Automates et programmation par contraintes pour la planification de personnel

Julien Menana

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

**ECOLE DOCTORALE SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATERIAUX**

Année 2011

N° attribué par la bibliothèque

Automates et programmation par contraintes pour la planification de personnel

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Programmation par contraintes

*Présentée
et soutenue publiquement par*

Julien Menana

*Le 28 octobre 2011 à l'École Nationale Supérieure
des Techniques Industrielles et des Mines de Nantes*

Devant le jury ci-dessous :

Rapporteurs	:	Jean-Charles Régin, Professeur	Université de Nice
		Louis-Martin Rousseau, Professeur	École Polytechnique de Montréal
Examinateurs	:	Benoît Rottembourg, Directeur de la recherche	EuroDécision Paris
		Patrice Boizumault, Professeur	Université de Caen
Directeur de thèse	:	Narendra Jussien, Professeur	École des Mines de Nantes
Responsable Scientifique	:	Sophie Demassey, Maître-assistante	École des Mines de Nantes

Équipe d'accueil : Contraintes, LINA UMR CNRS 6241
Laboratoire d'accueil : Département Informatique de l'École des Mines de Nantes
La Chantrerie – 4, rue Alfred Kastler – 44 307 Nantes

Automates et programmation par contraintes pour la planification de personnel

Automata and Constraint Programming for Personnel Scheduling Problems

Julien Menana

Remerciements

La thèse est une belle aventure, on passe par tous les états avant d'enfin entendre le jury vous déclarer Docteur. Si ce travail fait de moi l'expert des automates multi-pondérés en programmation par contraintes, je n'aurais pas pu le réaliser sans ce grand nombre de personnes qui m'ont entouré.

Une thèse, c'est une aventure en famille...

À Élodie qui en plus d'être la compagne idéale m'a toujours soutenu pendant cette thèse. Je n'aurais jamais pu terminer sans elle qui a sans doute lu ma thèse plus souvent que moi. J'espère que la nouvelle aventure dans laquelle nous nous sommes lancés connaîtra autant de succès !

À Chantal et Michel, mes parents, peu enthousiastes à l'idée de cette thèse, mais toujours présents dans les moments difficiles, notamment les corrections de « fotes d'aurtaugrafe ». Aussi, « si et seulement si » n'est pas une forme lourde de « si ».

À Guy, mon grand-père, pour l'expression de sa fierté lors de ma soutenance.

À Carine, ma sœur, Docteur en géographie, pour son soutien inconditionnel et sa compréhension des difficultés de la thèse.

À Soren et Célia, mon neveu et ma nièce, qui ne m'ont jamais demandé où j'en étais dans ma rédaction.

Mais il y a aussi les encadrants...

À Sophie, tout d'abord pour m'avoir supporté pendant ces 4 ans, mais surtout pour ses compétences et son intuition scientifique. Aussi bien son recul que nos « engueulades » m'ont permis plus que de finir ma thèse : de grandir. Je ne devrais pas l'avouer mais elle a eu raison sur beaucoup de choses. Bonne chance pour sa grande aventure également.

À Naren, pour son « pouvoir de nuisance » qui m'a permis de finir ma thèse. Non je plaisante, merci pour ses conseils toujours avisés, son amitié et son rire qui permet de dédramatiser beaucoup de situations.

Et les amis, les collègues, parfois les deux à la fois...

À Xavier et Bertille, pour leur amitié, leur canapé, leur télé les jours de match. Et surtout pour le modèle qu'ils représentent pour moi.

À Paula parce qu'elle a plus la « loose » que moi et que ça fait plaisir. Notez que je n'en rajoute pas puisqu'elle a osé soutenir sa thèse de médecine avant moi.

À Étienne, Julie, Céline, Sylvie et Xavier pour compléter la bande avec laquelle nous avons eu beaucoup de discussions enrichissantes et amusantes.

À Thierry avec qui j'ai passé quelques longues soirées à parler de tout.

À Philippe pour sa bonne humeur communicative.

À Fabien pour sa folie communicative.

À Maryannick pour avoir veillé sur moi pendant presque 10 ans à l'École des Mines.
À Sarah pour ses encouragements, et d'avoir supporté mes quolibets sur les sciences « molles ».

Enfin, je tiens à remercier le personnel et les enseignant-chercheurs de l'École des Mines sans qui rien n'aurait été possible.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Les problèmes de planification de personnel, du fait de leur complexité et de l'intérêt qu'ils suscitent au sein des milieux hospitalier comme industriel ont été beaucoup étudiés. Dès lors qu'une structure est organisée, la capacité de placer les bonnes personnes au bon moment devient cruciale pour satisfaire les besoins d'un service, d'une école ou d'une entreprise. À l'heure où les effectifs dans les hôpitaux sont de plus en plus restreints, construire de manière optimisée les emplois du temps de travail du personnel est un enjeu majeur pour assurer la qualité des soins. Pour une entreprise, dans un monde de plus en plus concurrentiel, utiliser au mieux ses ressources de personnel, donne un avantage certain à celle-ci. On citera notamment la planification du personnel navigant dans les compagnies aériennes qui, du fait des règles nombreuses et hétérogènes de la profession, constitue un problème extrêmement compliqué.

1.1 Objectifs

Si de nombreuses solutions logicielles existent pour la planification de personnel, la plupart sont basées sur des modèles figés, correspondant à une application donnée. L'approche que nous souhaitons développer dans cette thèse est de se rapprocher au plus du graal de la programmation par contraintes, c'est-à-dire d'exprimer un problème de manière simple et de ne pas intervenir dans le processus de résolution. Pour ce faire nous proposons de développer un framework intégrant à la fois des techniques de modélisation automatisées de règles de séquencement complexes, mais aussi des algorithmes efficaces de filtrage pour ces règles, élément essentiel du paradigme des contraintes. Ainsi, la résolution d'un problème de planification de personnel pourra être vu comme l'expression d'un ensemble de primitives de haut niveau.

1.2 Méthodologie de l'approche

Ce document couvre aussi bien les aspects de modélisation systématique des règles de séquencement sous forme d'automates que la mise en œuvre d'algorithmes de filtrage. Il est organisé au travers des trois parties suivantes :

Cadre général. La première partie est consacrée à la mise en place du cadre de notre étude. Nous rappellerons tout d'abord les éléments nécessaires à la compréhension de deux domaines : la programmation par contraintes et la théorie des langages. Nous discuterons ensuite des travaux existants liant ces deux domaines.

Modélisation automatique et filtrage de règles de séquencement. La seconde partie rassemblera quant à elle dans un premier temps nos contributions relatives à la modélisation automatique de règles de séquencement à l'aide de la théorie des langages. Le caractère expressif et compact des automates sera utilisé pour modéliser des règles de séquencement complexes. L'idée est de considérer une règle de séquencement comme un langage. Un séquencement valide par rapport à cette règle est alors un mot dudit langage. Les travaux existants proposent déjà de modéliser des règles de séquencement à l'aide de grammaires rationnelles voire hors contextes. À chacune de ces grammaires correspond un type d'automate. Nous proposons d'étendre ces champs en modélisant à l'aide d'automates pondérés et multi-pondérés. Cette extension nous permettra de modéliser des règles plus complexes de manière plus compacte. Ensuite, nous discuterons de l'opportunité d'agréger différentes règles qu'offre la stabilité des langages rationnels. Nous proposerons un algorithme permettant de réaliser l'intersection de deux règles représentées par des automates pondérés. Afin d'intégrer ces automates dans un modèle de programmation par contraintes, nous proposerons enfin un nouvel algorithme de filtrage basé sur la relaxation lagrangienne pour une contrainte à automate multi-pondéré : `multicost-regular`. La contrainte assure que les valeurs prises par une séquence de variables forment un mot accepté par un automate multi-pondéré. Nous proposons également une version souple de cette contrainte permettant de pénaliser les violations d'une règle qui serait modélisée par un tel automate : `soft-multicost-regular`.

Mise en œuvre pour la planification de personnel. Dans la troisième partie nous présenterons l'application principale de nos travaux : la planification de personnel. Nous proposerons par la suite un modèle basé sur les contraintes `multicost-regular` et `soft-multicost-regular` pour la résolution des problèmes de planification de personnel. Nous détaillerons nos différentes approches pour améliorer la résolution : nous discuterons de l'opportunité de relâcher les contraintes de couverture afin de résoudre le problème à l'aide de la relaxation lagrangienne. Nous proposerons également d'utiliser la structure des contraintes `multicost-regular` et `soft-multicost-regular` pour guider des heuristiques de recherche dédiées à ces problématiques. Enfin, nous détaillerons les expérimentations que nous avons faites pour mettre à l'épreuve notre framework sur les instances ASAP et NRP10.

1.3 Diffusion scientifique

Les différents travaux présentés dans ce document ont fait l'objet de diverses publications listées ci-dessous.

Conférence internationale avec comité de lecture

[1] Julien Menana and Sophie Demassey. Sequencing and Counting with the `multicost-regular` constraint. *6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09). Volume 5547 of Lecture Notes in Computer Science., Pittsburgh, USA, Springer-Verlag (May 2009) 178-192.*

[2] Julien Menana and Sophie Demassey. Weighted Automata, Constraint Programming and Large Neighborhood Search for the PATAT 2010 - NRP Competition. *Practice and Theory of Automated Timetabling, NRC 2010, Belfast, Northern Ireland (August 2010).*

Conférence nationale avec comité de lecture

[3] Julien Menana, Sophie Demassey and Narendra Jussien. Relaxation lagrangienne pour le filtrage d'une contrainte-automate à coûts multiples. $10^{ème}$ *conférence de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF 2009), Nancy (Février 2009).*

[4] Julien Menana, Sophie Demassey and Narendra Jussien. Séquencer et compter avec la contrainte `multicost-regular`. $5^{ème}$ *Journées Francophones de Programmation par contraintes (JFPC 2009), Orléans (Juin 2009).*

Rapport de recherche

[5] Julien Menana and Sophie Demassey. Modélisation et optimisation des préférences en planification de personnel. *Technical Report 11-01-INFO, École des Mines de Nantes.*

Première partie

Préliminaires

Chapitre 2

Introduction à la programmation par contraintes

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming : the user states the problem, the computer solves it. (E. Freuder)

Sommaire

La résolution de problèmes de planification de personnel que nous allons traiter dans cette thèse fait appel à la programmation par contraintes. Si les algorithmes que nous développerons sont complexes, les principes de base de la programmation par contraintes sont très abordables. Nous allons décrire dans ce chapitre les concepts de contrainte, filtrage, propagation et recherche de solution. Ces concepts et les notations associées seront utilisés dans la suite de cette thèse.

2.1 Réseau de contraintes

Un problème en programmation par contraintes, se modélise par un réseau de contraintes. Il se caractérise par un ensemble de variables, un ensemble de domaines et un ensemble de contraintes. À chaque variable est associé un domaine. Un domaine est un ensemble de valeurs pouvant être affectées à la variable associée. Une variable est dite instanciée lorsque son domaine est réduit à une seule valeur. Une instanciation est un ensemble de variables instanciées. Une contrainte est une relation sur un sous-ensemble de variables. Elle définit une ou plusieurs propriétés que l'instanciation simultanée des variables doit satisfaire. Chaque contrainte est donc en soi un sous-problème sur le sous-ensemble de ces variables support. Il existe divers types de contraintes, des plus simples, telle la relation binaire d'égalité, aux beaucoup plus complexes, telle la contrainte `tree` [15] qui modélise le partitionnement d'un graphe en arbres. Une solution d'un réseau de contraintes est une instanciation de toutes les variables satisfaisant la conjonction

de toutes les contraintes. Décider de l'existence d'une solution est en général un problème NP-Complet. Le problème de satisfaction de contraintes (CSP) où l'on cherche à exhiber une solution, si elle existe, ou à prouver l'irréalisabilité sinon, est un problème NP-Difficile.

Plus formellement, en reprenant la définition de Bessière [83, 20], un réseau de contraintes N est défini par :

- un ensemble fini de variables entières $\mathcal{X} = (x_1, \ldots, x_n)$;
- un domaine pour $\mathcal{X}$, c'est à dire un ensemble $\mathcal{D} = \mathcal{D}(x_1) \times \mathcal{D}(x_2) \times \ldots \times \mathcal{D}(x_n)$, tel que le domaine de chaque variable $x_i,\ \mathcal{D}(x_i) \subset \mathbb{Z}$;
- un ensemble de contraintes $\mathcal{C} = \{c_1, \ldots, c_e\}$, chacune portant sur un sous-ensemble $\mathcal{X}(c_j)$ de variable $\mathcal{X}$.

2.2 Algorithme backtrack

La recherche d'une solution peut se faire à l'aide d'algorithmes basés sur l'algorithme backtrack [50]. Les variables du problème sont instanciées dans un ordre donné jusqu'à ce qu'une (ou éventuellement plusieurs) contrainte ne soit plus satisfaite. On revient alors sur le dernier choix d'instanciation (backtrack) et on teste une autre valeur pour la variable courante. Si toutes les valeurs du domaine de cette variable ont été testées, sans succès, on réalise un backtrack sur la variable précédant la variable courante. Ce processus est répété jusqu'à ce que, soit une instanciation complète satisfaisant toutes les contraintes soit trouvée, soit un backtrack est réalisé à la racine de l'arbre de recherche. Si tout l'arbre de recherche est parcouru sans succès, le problème n'a pas de solution.

L'exemple suivant illustre la recherche d'une solution avec l'algorithme backtrack :

Exemple 1 (backtrack) *Considérons le réseau de contraintes suivant :*

- *soient x_1, x_2 et x_3 trois variables de domaines $D_1 = D_2 = D_3 = \{1, 2, 3\}$;*
- *les contraintes $C_1 = [x_1 \neq x_2]$ et $C_2 = [x_2 = x_3]$ sont posées.*

Nous supposons que les variables sont instanciées dans l'ordre x_1, x_2 puis x_3 en choisissant la plus petite valeur des domaines en premier. L'arbre de recherche parcouru avec l'algorithme backtrack est illustré figure 2.1.

2.3 Filtrage

L'exemple de la figure 2.1 montre les lacunes de l'algorithme backtrack. Pour réduire la taille de l'arbre de recherche en détectant au plus tôt l'irréalisabilité, une méthode de déduction est associée à chaque contrainte : il s'agit de son algorithme de filtrage. Un tel algorithme identifie et supprime les valeurs des domaines des variables impliquées dans la contrainte qui n'entrent dans aucune solution de la contrainte. Supprimer ces valeurs permet d'éviter de parcourir des branches de l'arbre qui ne peuvent aboutir à une solution. On parle de réduire l'espace de recherche. En programmation par contraintes, réduire l'espace de recherche à l'aide d'algorithmes de filtrage efficaces est une condition nécessaire à la résolution de problèmes complexes.

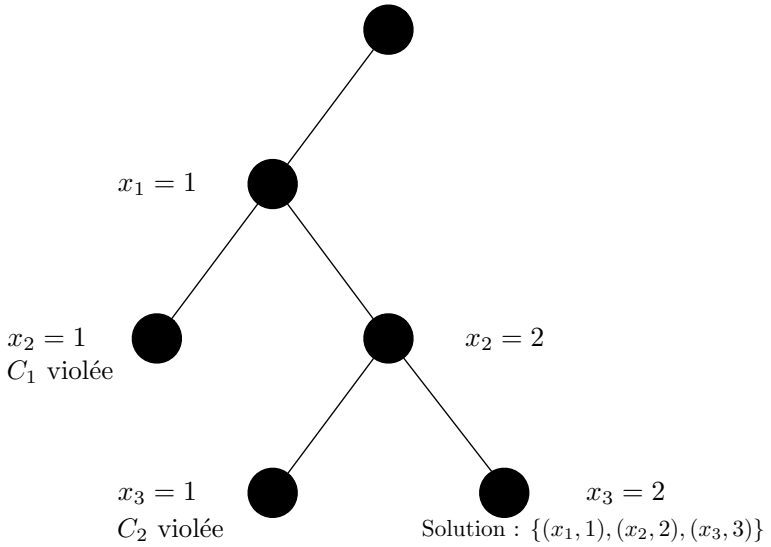


FIGURE 2.1 – Arbre de recherche construit par backtrack

Reprenons l'exemple de la section 2.2. Une fois posé $x_1 = 1$, l'algorithme de filtrage de la contrainte C_1 peut retirer la valeur 1 du domaine de x_2. En effet, la contrainte impose $x_1 \neq x_2$ et $x_1 = 1$, par conséquent, $x_2 \neq 1$. Cette déduction permet de ne pas tenter le choix $x_2 = 1$. Ainsi, la première valeur choisie par l'algorithme pour x_2 sera 2. $x_2 = 2$ étant posé, nous pouvons ici directement déduire la valeur de x_3. L'algorithme de filtrage de la contrainte C_2 maintient l'égalité entre les domaines de x_2 et x_3. x_2 étant instanciée, x_3 le devient automatiquement. Nous avons ainsi trouvé une solution à ce petit problème en explorant uniquement deux nœuds dans l'arbre de recherche contre cinq sans algorithme de filtrage. Si sur cet exemple le nombre de nœuds n'est pas critique, il le devient vite lorsque la taille des problèmes augmente. À noter que l'algorithme de filtrage engendre un coût supplémentaire en temps de calcul à chaque nœud de l'arbre. Il est donc important de mettre en balance la réduction de l'espace de recherche par le filtrage d'une contrainte et la complexité temporelle de cet algorithme.

2.4 Propagation de contraintes

Lorsque le domaine d'une variable est réduit, il faut réétudier l'ensemble des contraintes dans lesquelles la variable est impliquée. En effet, une modification d'un domaine pour une contrainte peut entraîner d'autres déductions de la part des algorithmes de filtrage d'autres contraintes. L'exemple suivant illustre ce mécanisme de propagation.

Exemple 2 (Propagation) *Prenons le problème de satisfaction de contraintes suivant :*

- *soient x_1, x_2 et x_3 trois variables de domaines $D_1 = D_2 = D_3 = \{1, 2, 3, 4\}$;*
- *on pose les contraintes $C_1 = [x_1 < x_2]$ et $C_2 = [x_2 < x_3]$.*

x_2 étant strictement supérieure à x_1, x_2 ne peut pas prendre la valeur 1. De manière symétrique x_1 ne peut être instanciée à 4. L'algorithme de filtrage de C_1 permet de déduire la réduction des domaines suivants :

$$D_1 = \{1, 2, 3\}, \text{ et } D_2 = \{2, 3, 4\}.$$

Si l'algorithme de filtrage de C_2 est appliqué à la suite, alors il déduit les réductions supplémentaires :

$$D_2 = \{2, 3\},\ \textit{et}\ D_3 = \{3, 4\}.$$

Le filtrage de C_2 ayant provoqué un nouveau changement dans le domaine de x_2, il se peut alors qu'appliquer le filtrage de C_1 permette de nouvelles déductions. En l'occurrence, celui-ci déduit maintenant :

$$D_1 = \{1, 2\}.$$

La propagation de contraintes consiste donc à appliquer les algorithmes de filtrage des contraintes dont les variables ont vu leur domaine réduit, à tour de rôle, jusqu'à ce que plus aucune réduction ne soit faite. On dit alors que le point fixe de la propagation est atteint.

2.5 Nature des contraintes

Les contraintes, dans leur prime définition, permettent de couvrir de nombreux aspects de modélisation. Il existe en réalité différents types de contraintes que nous caractérisons brièvement ici. Tout d'abord une contrainte peut être exprimée en extension comme l'ensemble explicite de ses solutions, sous forme d'un ensemble de n-uplets autorisés. Le plus généralement, une contrainte est définie en intention, de manière implicite. Par exemple, la contrainte $c_1(x_1, x_2) = [x_1 \neq x_2]$ est définie en intention : elle représente l'ensemble des couples de $\mathbb{Z}^2$ de valeurs différentes. La contrainte en intention $c_2(x_1, x_2) = \{(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)\}$ est équivalente à c_1 sur le domaine $\mathcal{D} = \{1,2,3\}x\{1,2,3\}$. Les contraintes en intention permettent de traduire des relations difficiles à expliciter, soit par le nombre de solutions trop important, soit parce que trouver une solution de la contrainte est difficile en soit.

L'arité d'une contrainte est le nombre de variables qu'elle implique. Une contrainte d'arité 2 est dite binaire. Une contrainte globale est une contrainte d'arité arbitraire définie par une formule. L'exemple de contrainte globale le plus courant est la contrainte `alldifferent` [79]. `alldifferent`$(x_1, \ldots, x_n) \equiv \bigwedge_{i \neq j}[x_i \neq x_j]$. L'intérêt des contraintes globales est double. Elles apportent d'une part un gain en terme d'expressivité. En effet, une contrainte étant un sous problème que l'on est capable d'identifier, il est souvent plus aisé de l'exprimer à l'aide de primitives plus globales. Par exemple, il est plus simple d'exprimer pour un ensemble de variables qu'elles doivent être différentes deux à deux que d'exprimer toutes les paires de variables pour poser des contraintes binaires. D'autre part, l'aspect global de ces contraintes permet de réaliser plus de déductions par filtrage. Ainsi la contrainte `alldifferent` permet de faire plus de filtrage qu'une conjonction de contraintes `neq`.

Exemple 3 *Regardons les déductions faites par propagation des contraintes sur le réseau suivant :*

- *soient x_1, x_2 et x_3, trois variables de domaines $\mathcal{D}(x_1) = \mathcal{D}(x_2) = \{1,2\}$ et $\mathcal{D}(x_3) = \{1,2,3\}$;*
- *et soient les contraintes $c_1 = [x_1 \neq x_2], c_2 = [x_1 \neq x_3]$ et $c_3 = [x_2 \neq x_3]$.*

Comme pour l'exemple précédent, nous appliquons les algorithmes de filtrage des contraintes jusqu'à l'obtention d'un point fixe. Or pour ce réseau, aucune déduction ne peut être faite. En effet, pour chaque variable, toute valeur de son domaine peut être étendue à une solution pour chaque contrainte dans laquelle elle est impliquée.

Posons maintenant, à la place de la conjonction de différence, la contrainte `alldifferent`(x_1, x_2, x_3). L'algorithme de filtrage de cette contrainte globale proposé par Régin [79] construit le graphe bipartite où

chaque arête représente une affectation possible pour une variable (figure 2.2). Ces couplages de cardinalité 3 dans ce graphe correspondent exactement aux solutions de la contrainte (en rouge sur le schéma). Par la suite, tous les arcs n'appartenant pas au couplage sont inversés. Ainsi tout arc n'appartenant ni au couplage, ni à aucune composante fortement connexe (en bleu dans le graphe) peut être supprimé.

Pour plus d'information à ce sujet, nous vous invitons à consulter l'article de Régin sur `alldifferent` *[79], ainsi que le livre de Berge sur la théorie des graphes [16].*

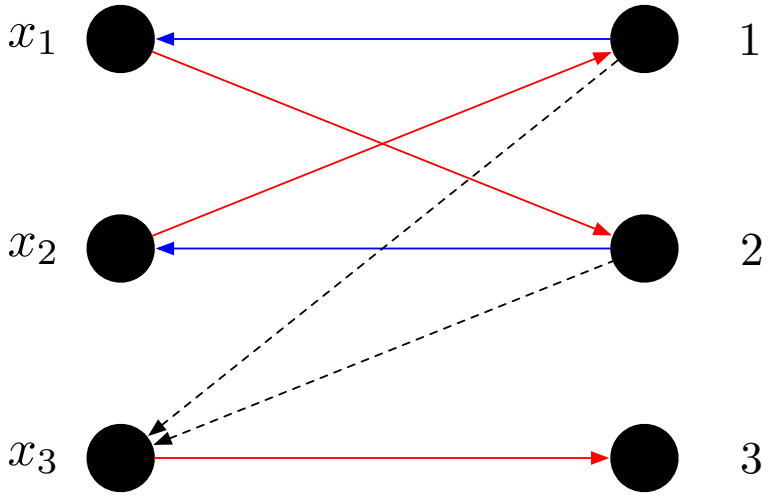


FIGURE 2.2 – Graphe bipartite pour la contrainte `alldifferent`

Appliquer l'algorithme de filtrage de la contrainte globale `alldifferent` *permet ainsi de déduire que* x_3 *ne peut prendre les valeurs* 1 *et* 2*, d'où* $x_3 = 3$.

Une modélisation de problèmes à l'aide de contraintes globales permet donc plus de filtrage et par conséquent, le plus souvent, une résolution plus rapide.

L'algorithme de la contrainte `alldifferent` permet d'atteindre ce que l'on appelle la cohérence d'arc (AC). Ce terme indique le niveau de filtrage de la contrainte.

Définition 1 (Cohérence d'arc [20]) *Soit un réseau* $N = (\mathcal{X}, \mathcal{D}, \mathcal{C})$*, une contrainte* $c \in \mathcal{C}$ *et une variable* $x_i \in \mathcal{X}(C)$

- *Une valeur* $v_i \in \mathcal{D}(x_i)$ *est cohérente par rapport à* c *dans* $\mathcal{D}$ *si et seulement si il existe un n-uplet autorisé contenant la valeur* v_i *pour la variable* x_i*. Ce n-uplet est appelé support du couple* (x_i, v_i) *dans* c.

- *Une contrainte* c *est dite arc-cohérente si et seulement si, toutes les valeurs des domaines des variables de* c *sont cohérentes par rapport à* c.

En d'autres termes, une contrainte ayant un algorithme de filtrage faisant la cohérence d'arc supprimera toute valeur d'un domaine qui ne peut être étendue à une solution.

Nous invitons le lecteur souhaitant en savoir plus sur les différentes cohérences à lire le rapport technique sur la propagation de contraintes de Bessière [20].

Un catalogue de contraintes globales [12] référence et décrit, à l'heure actuelle, plus de 300 contraintes globales.

2.6 Heuristiques de branchement

La construction de l'arbre de recherche repose sur l'ordre de choix des variables et des valeurs d'instanciation. Dans l'exemple 2.2, les variables sont sélectionnées par ordre lexicographique. L'efficacité de la résolution ne dépend pas uniquement de la qualité du filtrage et de la propagation. Les stratégies de sélection de la prochaine variable ou de la prochaine valeur à choisir jouent un rôle important.

Il est souvent utile, parfois nécessaire, de définir des stratégies de recherche spécifiques au problème que l'on traite afin d'améliorer la recherche de solutions. Cependant, il existe des stratégies génériques qui s'appliquent à tous les problèmes. Une des plus usitées consiste à sélectionner la variable de plus petit domaine [52]. Des heuristiques génériques plus sophistiquées ont fait leur apparition telle l'heuristique ordonnant dynamiquement les variables en fonction du nombre de contraintes dans lesquelles elles sont impliquées [27]. Des heuristiques utilisant le ratio entre degré de la variable et taille des domaines ont montré d'excellents résultats [19]. Enfin les dernières heuristiques générales apparues utilisent la notion d'impact, mesure de la performance d'une affectation donnée [76].

2.7 Conclusion

La programmation par contraintes est donc un cadre qui fournit des outils de modélisation et de résolution de problèmes combinatoires. Un problème se modélise au moyen de variables avec leur domaine, et de contraintes, relations logiques entre les variables. La recherche de solutions est effectuée à l'aide de l'algorithme backtrack, amélioré de la propagation des contraintes. Pour un problème donné, il existe toujours différents modèles possibles pour une application donnée, en terme de variables, de contraintes et de stratégies de recherche. L'efficacité de la résolution dépend largement du modèle choisi. Ainsi, modéliser un problème nécessite deux phases importantes :

- exprimer le problème à l'aide d'un bon modèle ;
- utiliser des algorithmes de filtrage efficaces.

L'utilisation de contraintes globales est indiquée pour modéliser et résoudre des problèmes complexes. Construire un modèle à base de contraintes globales offre une approche plus naturelle de la modélisation. En revanche, construire les structures de données utilisées par ces contraintes n'est pas toujours simple : par exemple, la contrainte `tree`, introduite au début de ce chapitre, nécessite un graphe [15].

Si la programmation par contraintes permet de décrire des problèmes combinatoires complexes grâce aux primitives de haut niveau que sont les contraintes globales, nous considérons que la modélisation à l'aide d'automates est une étape supplémentaire vers la création d'un outil générique et de haut niveau pour la résolution de problèmes. Les concepts de la théorie des langages et des automates seront ainsi introduits au chapitre suivant.

Chapitre 3

Langages et automates

Tous les moyens de l'esprit sont enfermés dans le langage ; et qui n'a point réfléchi sur le langage n'a point réfléchi du tout. (Alain, dans Propos sur l'éducation, 1932)

Sommaire

Nous nous intéresserons dans ce chapitre à la théorie des langages et des automates. Ce formalisme, introduit par Noam Chomsky [37, 38] permet d'exprimer de manière générative des ensembles. Nous allons d'abord détailler ce qu'est une grammaire formelle. Puis nous nous intéresserons plus particulièrement aux grammaires rationnelles. Une des propriétés de ces grammaires est de générer des langages reconnaissables par des automates finis. Nous rappellerons les concepts de base des automates finis. Enfin nous terminerons par une introduction aux automates pondérés que nous utiliserons intensément dans les prochains chapitres. L'essentiel des définitions présentes dans ce chapitre sont issues des livres *Introduction to Automata Theory, Languages, and Computation* d'Hopcroft, Motwani et Ullman [53], et *Élement de théorie des automates* de Jacques Sakarovitch [84].

3.1 Grammaires formelles

Les grammaires formelles ou grammaires génératives sont issues des travaux de Chomsky [37] en linguistique. L'objectif de Chomsky était de rendre compte des structures innées de la faculté du langage. Aujourd'hui, les grammaires formelles sont toujours utilisées dans des domaines aussi variés que le traitement du langage naturel, la conception d'interprètes et compilateurs, l'analyse de programmes ou la conception d'ordonnanceurs.

Nous nous intéresserons dans cette section d'abord au formalisme utilisé par les grammaires formelles, puis nous présenterons la hiérarchie de Chomsky permettant de caractériser différents niveaux de grammaires selon leur expressivité.

3.1.1 Vocabulaire

Les grammaires formelles sont utilisées pour définir et générer des langages.

Définition 2 (Alphabet) *Un alphabet est un ensemble fini ou non de symboles. Par exemple,* $\{a, b, \ldots, z\}$ *est l'alphabet de la langue anglaise.*

Définition 3 (Mot) *Un mot* m *sur un alphabet* Σ *est une séquence finie de symboles issus de* Σ*.*

Définition 4 (Langage) *Un langage est un ensemble de mots formés par des séquences de symboles issus d'un alphabet.*

Définition 5 (Fermeture de Kleene) *La fermeture de Kleene, est l'opérateur unaire qui, appliqué à un ensemble* V*, forme le langage* $V*$*.*

- *Si* V *est un alphabet (i.e. un ensemble de symboles), alors* $V*$ *est l'ensemble des mots sur* V*, le mot vide* ϵ *inclus.*
- *Si* V *est un langage (i.e. un ensemble de mots), alors* V^* *est le plus petit langage qui le contienne, qui contienne* ϵ *et qui soit stable par concaténation.*

Cet opérateur permet, étant donné un ensemble de mots ou de symboles, de créer de nouveaux mots par concaténation des primitives fournies par l'ensemble de départ. Il permet par exemple d'indiquer qu'un motif doit pouvoir se répéter.

3.1.2 Formalisme des grammaires

Une grammaire formelle décrit la manière de générer des mots appartenant à un langage.
Voici la définition formelle des grammaires génératives proposée par Noam Chomsky [37, 38] :

Définition 6 (Grammaire) *Une grammaire* $\mathcal{G}$ *est un n-uplet* (N, Σ, P, S) *où :*

- N *est un ensemble fini de symboles non terminaux (qui n'apparaissent pas dans les mots) ;*
- Σ *est un ensemble fini de symboles terminaux (qui constituent les mots) ;*
- *un ensemble fini* P*, de règles de production. Une règle de production, notée* $S_1 \to S_2$ *associe l'élément* $S_1 \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ *à un élément* $S_2 \in (\Sigma \cup N)^*$ *;*
- *un symbole* $S \in N$*, symbole de départ.*

Définition 7 (Dérivation) *Soit une grammaire* $\mathcal{G} = (N, \Sigma, P, S)$*. On appelle dérivation le fait d'appliquer une règle de production au symbole de départ ou à un mot du langage. Pour tout* $a, b \in (\Sigma \cup N)^*$ *On notera* $aS_1b \Rightarrow_{\mathcal{G}} aS_2b$ *si et seulement si la règle* $S_1 \to S_2$ *est dans* P*. On notera* $aS_1b \Rightarrow^*_{\mathcal{G}} aS_nb$ *s'il existe une chaîne de règles dans* P *permettant de transformer* S_1 *en* S_n *(*$S_1 \to S_2, S_2 \to S_3, \ldots, S_{n-1} \to S_n$*).*

Exemple 4 *Nous définissons ici la grammaire* $\mathcal{G} = (N, \Sigma, P, S)$ *permettant de construire les mots de la forme* $a^n b^n$ *pour* $n \in \mathbb{N}$ *(*a^n *signifie* n *répétition de* a*, e.g.* $a^3 = aaa$*). On pose :*

- $N = \{S\}$;
- $\Sigma = \{a, b\}$;
- $P = \{S \to aSb, S \to \epsilon\}$.

Ainsi, pour construire le mot aaabbb à partir de la grammaire $\mathcal{G}$, nous dérivons trois fois le symbole non-terminal S à l'aide de la première règle de production et une fois à l'aide de la seconde. Ce qui donne :

- $S \Rightarrow_{\mathcal{G}} aSb$ *(règle 1)* ;
- $aSb \Rightarrow_{\mathcal{G}} aaSbb$ *(règle 1)* ;
- $aaSbb \Rightarrow_{\mathcal{G}} aaaSbbb$ *(règle 1)* ;
- $aaaSbbb \Rightarrow_{\mathcal{G}} aaa\epsilon bbb = aaabbb$ *(règle 2).*

Selon les caractéristiques de l'ensemble de règles de production P, il existe une hiérarchisation des grammaires formelles, connue sous le nom de hiérarchie de Chomsky.

3.1.3 Hiérarchie de Chomsky

La hiérarchie de Chomsky permet de caractériser les grammaires en fonction de la forme de leurs règles de production. Celles-ci reflètent l'expressivité de la grammaire.

Définition 8 (Problème du mot) *Soit une grammaire $\mathcal{G} = (N, \Sigma, P, S)$ et un mot $m \in \Sigma*$. Le problème du mot consiste à décider si le mot m appartient ou non au langage $\mathcal{L}_{\mathcal{G}}$ défini par la grammaire $\mathcal{G}$.*

La décidabilité et la complexité du problème du mot pour un langage dépendent du type de la grammaire qui a généré ce langage. On différenciera principalement quatre types de grammaires :

- **Grammaires de type 0 :** elles incluent toutes les grammaires formelles. Le problème du mot sur cette classe de grammaire est indécidable.
- **Grammaires de type 1 :** appelées également grammaires contextuelles, ces grammaires ont des règles de la forme $aSb \to ayb$ où S est un symbole non-terminal, a, b et y des chaînes composées de symboles terminaux et non-terminaux. y ne peut pas être la chaîne vide. Le problème de l'appartenance d'un mot est ici décidable mais PSPACE-complet [56].
- **Grammaires de type 2 :** autrement appelées grammaires non-contextuelles, les règles de production sont de la forme $S \to y$ où S est un symbole non-terminal et y une chaîne composée de symboles de tous types. Le problème du mot est résolu en temps polynomial en fonction de la taille du mot d'entrée.
- **Grammaires de type 3 :** ou grammaires rationnelles. Elles génèrent les langages rationnels. Les règles sont limitées à un symbole non terminal dans la partie gauche de la règle et, dans la partie droite, à une chaîne de symboles terminaux suivis ou précédés d'un symbole non-terminal. Le problème du mot se fait en temps linéaire par rapport à la taille du mot.

Notons que chaque type de grammaire est inclus dans le type précédent.

Chacune de ces classes de grammaire possède une machine ou un automate équivalent. Ainsi, les langages générés par les grammaires formelles sont reconnus par une machine de Turing [91]. C'est-à-dire que pour tout mot d'un langage généré par une grammaire de type 0, il existe une machine de Turing qui acceptera le mot en temps fini. Le problème du mot n'est pas décidable car il est possible que la machine de Turing ne réponde jamais si le mot n'appartient pas au langage. Les langages décrits par les grammaires de type 1 sont reconnus par une machine de Turing dont la taille du ruban est bornée par un facteur de la taille du mot à reconnaître.

Les langages définis par des grammaires de type 2 ou 3 sont reconnus par des machines particulières : les automates à piles pour les premières et les automates finis pour les secondes. Cette dernière classe d'automates sera le sujet principal de notre étude.

Le tableau 3.1 résume les différents types de grammaires.

Type	Machine ou automate associé	Complexité du problème du mot
Toutes formelles	Machine de Turing	Indécidable
Contextuelle	Machine de Turing à ruban borné	`PSPACE-Complet`
Non-contextuelle	Automate à piles	$O(n^3)$
Rationnelle	Automate fini	$O(n)$

TABLE 3.1 – Grammaires et automates associés

Pour des raisons de complexité, nous ne parlerons maintenant que des grammaires de types 2 et 3. En effet, nous souhaitons utiliser la puissance de la théorie des langages pour modéliser des règles issues des problèmes de planification de personnel. Kadioglu et Sellmann [56] ont discuté l'utilisation de grammaires pour la modélisation de problèmes de séquencement et ont montré qu'il était très difficile et inutile de modéliser des problèmes à l'aide de grammaires plus complexes que les non-contextuelles.

3.2 Automates finis

Il existe une équivalence entre les grammaires et certaines classes d'automates. Un automate est un modèle abstrait de machine qui lit des séquences de symboles. L'ensemble des séquences lues par un automate est le langage reconnu par cet automate. Le théorème de Kleene [58] montre ainsi que les grammaires rationnelles engendrent des langages reconnus par des automates finis. Les grammaires non contextuelles génèrent des langages reconnus par des automates à piles.

Mais qu'est-ce qu'un automate ? Une première définition est donnée par Sakarovitch [84] : « Un automate est un graphe, orienté et étiqueté par les lettres d'un alphabet, et dans lequel deux sous-ensembles de sommets ont été distingués. »

Nous distinguerons dans cette section trois types d'automates : les automates finis simples, les expressions rationnelles et les automates pondérés.

3.2.1 Automates finis simples

Définition 9 (Automate fini) *Un automate fini $\mathcal{A}$ est un quintuplet (Q, Σ, E, I, T) où :*

- *Q est un ensemble non vide, appelé ensemble des états de $\mathcal{A}$;*
- *Σ est un ensemble non vide, appelé l'alphabet de $\mathcal{A}$;*

- *I et T sont deux sous-ensembles de Q ; I est l'ensemble des états initiaux et T est l'ensemble des états finals de $\mathcal{A}$;*
- *E est un sous-ensemble de $Q \times \Sigma \times Q$, appelé ensemble des transitions de $\mathcal{A}$.*

Définition 10 (Chemin dans un automate fini) *Un chemin π dans un automate fini est une suite de transitions e_1 e_2 e_k $e_n \in E^*$ consécutives. C'est-à-dire, pour tout $i \in [\![1, n-1]\!]$, l'état final de la transition e_i est l'état initial de la transition e_{i+1}. Pour toute transition $e \in E$, on notera :*

- *$orig(e)$, l'état initial de la transition ;*
- *$dest(e)$, l'état final de la transition ;*
- *$symb(e)$, le symbole porté par la transition.*

Définition 11 (Reconnaissance d'un mot par un automate) *Un mot m sur l'alphabet Σ est reconnu par un automate $\mathcal{A} = (Q, \Sigma, E, I, T)$, s'il existe un chemin de I vers T tel que la séquence des étiquettes des arcs du chemin forme le mot. On dit également que l'automate accepte le mot m.*

L'ensemble des mots reconnus par un automate forme le langage reconnu par cet automate.

Exemple 5 *Soit l'automate $\mathcal{A} = (Q, \Sigma, E, I, T)$ défini de la manière suivante :*

- *$Q = \{q_0, q_1, q_2\}$;*
- *$\Sigma = \{a, b\}$;*
- *$E = \{(q_0, a, q_1), (q_1, a, q_2), (q_2, a, q_2)(q_2, b, q_2)\}$;*
- *$I = \{q_0\}$;*
- *$T = \{q_2\}$.*

Cet automate accepte tous les mots formés par les symboles a et b préfixés par le mot aa. La figure 3.1 présente l'automate sous la forme d'un graphe.

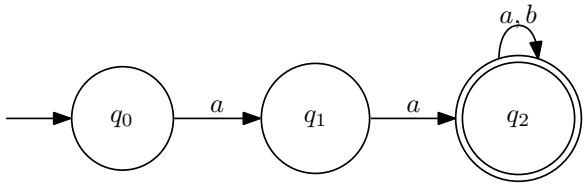


FIGURE 3.1 – Automate représentant le langage des mots préfixés par aa

Les automates finis sont répartis en deux classes, les automates finis déterministes (AFD) et les automates finis non-déterministes (AFN). La différence est dans l'impossibilité pour un AFD d'avoir, depuis un état, plus d'une transition étiquetée par un même symbole. De plus, l'ensemble des états initiaux doit être composé d'un seul état. Nous pouvons donc définir un AFD de la manière suivante :

Définition 12 (Automate fini déterministe) *Un automate fini déterministe $\mathcal{A}_D$ est un automate fini où :*

- *I est composé d'un état unique i ;*
- *il n'existe pas deux transitions de même valeur issues d'un même état.*

Dans le cas d'un AFD, l'ensemble des transitions de l'automate peut être défini comme une fonction de transition $\delta : M_\delta \subset Q \times \Sigma \to Q$ avec $\delta(q_i, \alpha) = q_j \iff (q_i, \alpha, q_j) \in E$. Notons que la fonction n'est pas forcément définie pour tous les états et pour tous les symboles si $M_\delta \subsetneq Q \times \Sigma$. On parle alors d'automate incomplet. Un automate incomplet peut être complété en ajoutant un état non final et toutes les transitions manquantes vers cet état. L'automate figure 3.1 remplit toutes les conditions pour être un automate fini déterministe.

Un automate fini non déterministe peut au contraire être « dans plusieurs états à la fois ». L'état initial n'est plus forcément unique puisque l'on peut cependant toujours fusionner tous les états initiaux dans un AFN.

Exemple 6 *Un exemple courant où il est facile d'écrire un AFN est lorsque nous souhaitons reconnaître un ensemble de mots dont on ne connaît que le suffixe. Soit l'automate $\mathcal{A}_N$ défini par :*

- $Q = \{q_0, q_1, q_2\}$;
- $\Sigma = \{0, 1\}$;
- $E = \{(q_0, 0, q_0), (q_0, 1, q_0), (q_0, 0, q_1), (q_1, 1, q_2)\}$;
- $s = q_0$;
- $T = \{q_2\}$.

L'existence des transitions $(q_0, 0, q_0)$ et $(q_0, 0, q_1)$ nous montre que cet automate est non déterministe. La construction de cet automate figure 3.2 permet plus facilement de saisir le langage qu'il définit. Ainsi, ne seront acceptés que les nombres binaires se terminant par 01.

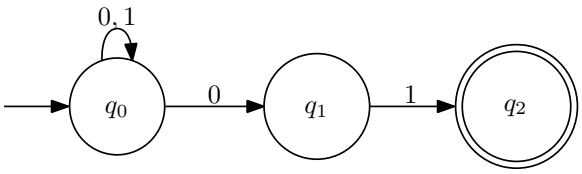


FIGURE 3.2 – AFN représentant le langage des nombres binaires finissant par 01

La question légitime qui vient après l'exemple 6 est la suivante : Existe-il un AFD équivalent, c'est-à-dire qui reconnait les nombres binaires suffixés par 01 ? Regardons l'automate défini dans l'exemple 7.

Exemple 7 *Soit $\mathcal{A}_D$ l'automate défini par :*

- $Q = \{q_0, q_1, q_2\}$;
- $\Sigma = \{0, 1\}$;
- $E = \{(q_0, 0, q_1), (q_0, 1, q_0), (q_1, 0, q_1), (q_1, 1, q_2), (q_2, 0, q_1), (q_2, 1, q_0)\}$;

- $s = q_0$;
- $T = \{q_2\}$.

Cet automate est déterministe. Pour arriver sur l'état q_2, seul état final, le mot lu par l'automate doit se terminer par 01*, ce qui décrit exactement le langage de l'exemple 6. La figure 3.3 illustre l'automate décrit ici.*

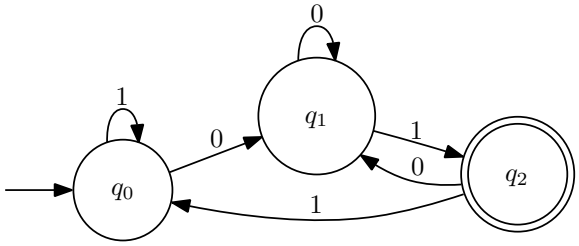


FIGURE 3.3 – AFD représentant le langage des nombres binaires finissant par 01

Il existe un processus de déterminisation qui transforme un AFN en AFD acceptant le même langage. Nous reviendrons dans le chapitre 6 sur l'algorithme réalisant cette transformation. Le langage défini dans les exemples 6 et 7 montre qu'il existe des langages où il est plus facile de construire un AFN qu'un AFD. Hopcroft et .al [53] postulent qu'en pratique un AFD construit à partir d'un AFN aura environ le même nombre d'états. Cependant, il arrive que le plus petit AFD construit à partir d'un AFN contienne jusqu'à $2^{|Q|}$ états. La preuve de l'équivalence entre AFD et AFN se fait par une méthode constructive où les états accessibles depuis un même état par une transition de même étiquette sont regroupés.

3.2.2 Expressions rationnelles

Quittons un instant le monde de la description de langages sous forme de machine. Les expressions rationnelles sont une description algébrique des langages rationnels. Appelées autrement motifs, elles décrivent de fait des ensembles de mots. L'utilisation couramment faite de cet outil est de décrire de manière concise des ensembles, sans avoir à lister tous les éléments de cet ensemble. Elles ont le même pouvoir expressif que les grammaires rationnelles. Une expression rationnelle est composée de constantes et d'opérateurs caractérisant des ensembles de mots et les opérations associées à ces mots.

Définition 13 (Expression rationnelle) *Soit Σ un alphabet, une expression rationnelle est composée des constantes :*

- *$\emptyset$, l'ensemble vide ;*
- *σ, le mot vide ;*
- *$a \in \Sigma$ un caractère.*

Et des opérations :

- **concaténation**, *soit un ensemble R et un ensemble S, RS représente l'ensemble des concaténations des mots de R et S. Par exemple $R = \{ab, c\}$ et $S : \{d, ef\}$, $RS = \{abd, abef, cd, cef\}$;*
- **alternation**, *étant donné un ensemble R et un ensemble S, $R|S$ est l'ensemble des mots de R et de S. Par exemple, pour les mêmes R et S, $R|S = \{ab, c, d, ef\}$;*

- **étoile de Kleene**, *soit R un ensemble, R^* dénote l'ensemble construit en répétant zéro (ou plus) fois les mots de R.*

Exemple 8 *Modéliser le langage des nombres binaires finissant par* 01 *se fait de la manière suivante à l'aide des expressions rationnelles :*

- *représenter un nombre binaire :* $(0|1)^*$. *Cette expression représente une séquence de* 0 *ou de* 1 *répétée zéro fois ou plus ;*
- *représenter le suffixe obligatoire :* 01 *;*
- *l'expression rationnelle représentant le langage des nombres binaires finissant par* 01 *se fait par concaténation des expressions précédentes :* $(0|1)^*01$.

L'exemple 8 illustre la simplicité d'utilisation des expressions rationnelles. L'expression $(0|1)^*01$ représente le même langage que les automates figures 3.2 et 3.3, mais de manière plus compacte. Les automates finis déterministes, les automates finis non déterministes et les expressions rationnelles (er) décrivent tous les trois des langages rationnels. L'équivalence entre ces trois représentations est démontrée notamment dans le chapitre 3 de *Introduction to Automata Theory, Languages and Computation* [53]. Elle est résumée par le diagramme suivant figure 3.4 qui montre les transformations possibles entre les expressions rationnelles et les différents automates finis[1].

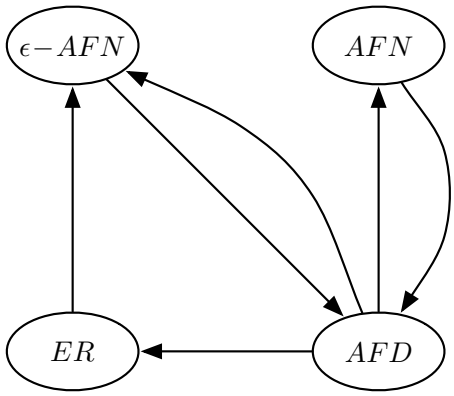


FIGURE 3.4 – Diagramme illustrant les équivalences entre les 4 notations possibles des langages rationnels.

3.2.3 Automates finis pondérés

Les automates finis, comme les expressions rationnelles, permettent de modéliser des langages rationnels. Afin de modéliser des langages plus complexes, dans le sens plus haut dans la hiérarchie de Chomsky, il existe d'autres formes de grammaires, représentées par d'autres types d'automates ou de machines. Il existe également une extension des automates finis où les transitions sont associées à un poids. Un mot du langage est alors une séquence de symboles et de poids.

Nous introduisons ici les automates pondérés qui décrivent une classe de langage strictement supérieure aux langages rationnels [60, 84].

La définition formelle d'un automate pondéré proposée par Kuich et Salomaa [60] est basée sur la notion de semi-anneau.

[1] Les ϵ-AFN sont une forme d'AFN possédant des transitions dites spontanées : facultatives et ne consommant pas de symbole. Nous n'utilisons pas directement ces automates.

Définition 14 (Automate pondéré) *Un automate pondéré est un quintuplet (Q, Σ, E, I, T), similaire à un automate fini où E, l'ensemble des transitions, est un sous-ensemble de $Q \times \Sigma \times Q \times \mathbb{K}$ tel que :*

- *$(\mathbb{K}, \oplus, \otimes)$ est un semi-anneau ;*
- *pour tout triplet $(q, \sigma, q') \in Q \times \Sigma \times Q$, il existe un et un seul élément $k \in \mathbb{K}$ tel que $(q, \sigma, q', k) \in E$.*

Définition 15 (Fonction de poids) *La fonction de poids est l'application w qui associe à toute paire $(q, \sigma) \in Q \times \Sigma$ un élément $k \in \mathbb{K}$.*

Définition 16 (Poids d'un chemin) *Soit un chemin $\pi = e_1 e_2 \dots e_n$, le poids du chemin $w(\pi)$ dans l'automate pondéré $\mathcal{A} = (Q, \Sigma, E, I, T)$ dans le semi-anneau $(\mathbb{K}, \oplus, \otimes)$, et le $\otimes$-produit des poids de toutes les transitions. On note :*

$$w(\pi) = w(orig(e_1), symb(e_1)) \otimes w(orig(e_2), symb(e_2)) \otimes \dots \otimes w(orig(e_n), symb(e_n)).$$

Définition 17 (Poids d'un mot) *Le poids d'un mot m pour un automate pondéré $\mathcal{A}$ dans le semi-anneau $(\mathbb{K}, \oplus, \otimes)$ est la $\oplus$-somme de tous les poids des chemins correspondant à m dans l'automate. On note :*

$$w(m) = \bigoplus_{\pi \in \Pi(m)} w(\pi).$$

Notons que si l'automate est déterministe, alors il n'existera qu'un seul chemin pour un mot donné, alors la première opération du semi-anneau ne sera jamais utilisée et le poids d'un mot m sera exactement le poids de son unique chemin.

L'acceptation d'un mot par un automate pondéré se fait par deux critères :

- Le mot est-il dans le langage défini par l'automate fini simple (i.e. un chemin de I vers T) ?
- Le poids du mot a-t-il une valeur donnée ?

Concernant la valeur admissible d'un mot, on définit souvent un sous-ensemble $\mathbb{J} \in \mathbb{K}$, valeurs du poids pour lesquelles un mot est admissible. Ainsi, un mot m est accepté dans un automate pondéré $\mathcal{A}$ dans le semi-anneau $(\mathbb{K}, \oplus, \otimes)$ si et seulement si :

- $m \in \mathcal{L}(\mathcal{A})$;
- $w(m) \in \mathbb{J}$.

Exemple 9 (Un automate pondéré comptant une séquence) *Soit le semi-anneau $(\mathbb{N}, \min, +)$ et l'automate $\mathcal{A} = (Q, \Sigma, E, I, T)$ avec :*

- *$Q = \{q_0, q_1, q_2\}$;*
- *$\Sigma = \{0, 1\}$;*
- *$E = \{(q_0, 1, q_1, 0), (q_0, 0, q_2, 0), (q_1, 1, q_1, 0), (q_1, 0, q_2, 0), (q_2, 0, q_2, 0), (q_2, 1, q_0, 1)\}$;*
- *$I = \{q_0\}$;*
- *$T = \{q_0\}$.*

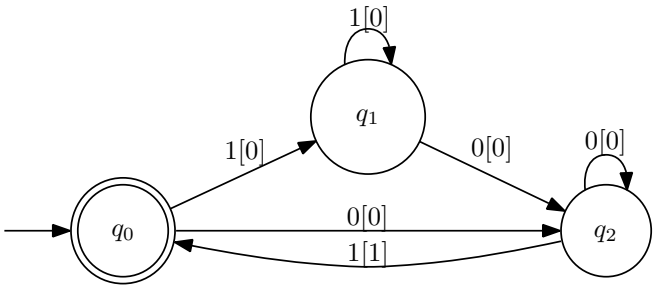


FIGURE 3.5 – Automate comptant les occurrences du motif 01

Cet automate est présenté figure 3.5. Tel quel, il accepte tous les mots de Σ^* *suffixés par* 01 *et compte le nombre de fois où le motif* 01 *apparaît dans un mot. Si on pose* $J = (\{2, 3, 4\}, \min, +) \subset (\mathbb{N}, \min, +)$*, alors l'automate pondéré figure 3.5 accepte tous les mots de* $\Sigma*$ *suffixés par* 01 *et tel que ce motif apparaisse entre 2 et 4 fois inclus.*

Pour en savoir plus sur les automates pondérés, nous invitons le lecteur à se tourner vers l'ouvrage *Handbook of weighted Automata* de Droste et al. [45] ou à lire le chapitre consacré aux $\mathbb{K}$-automates du livre *Éléments de théorie des automates* de Jacques Sakarovitch [84]. Mohri et al. [68] présentent une application des automates pondérés dans le cadre de la reconnaissance vocale. Jiang et al. [55] quant à eux utilisent les automates pondérés pour la compression d'images. Ils utilisent un semi-anneau et un automate pour associer chaque séquence reconnaissable à un nombre unique.

3.3 Conclusion

Nous avons présenté dans ce chapitre une petite partie de la théorie des langages et des automates. Afin d'introduire les langages rationnels et leurs représentations sous forme d'automates finis ou d'expressions rationnelles, nous avons introduit les grammaires formelles présentées originellement par Chomsky. Dans le prochain chapitre, nous vous présenterons les origines et les utilisations des automates en programmation par contraintes de la littérature.

Chapitre 4

Automates en programmation par contraintes

Sommaire

DANS les chapitres précédents, nous avons introduit la programmation par contraintes et les langages formels. Dans ce chapitre, nous décrirons le lien qui existe entre ces deux disciplines. Nous présenterons l'origine de l'utilisation des automates en programmation par contraintes jusqu'aux derniers travaux sur le sujet. Les contraintes connexes aux contraintes automates seront également présentées.

4.1 Origines

L'origine de l'utilisation des automates en programmation par contraintes est multiple. Une des premières apparitions de l'utilisation d'automates dans un système de contraintes est proposée par Schulz et al. [85]. Dans le cadre de la reconnaissance de phrases dans des corpus de grande taille, ils proposent d'utiliser la structure compacte des automates pour représenter des structures grammaticales complexes. Ils utilisent ensuite la propagation de contraintes pour s'assurer de certaines propriétés morphologiques.

Si la combinaison des automates et des contraintes est présente dans cet article, il n'apparaît pas encore de contrainte automate à proprement parler. Le cheminement vers un algorithme de filtrage pour une contrainte portant sur une séquence de variables, se devant de respecter un motif donné, s'est fait au travers d'applications de planification de personnel. Ces dernières ont conduit Pesant à développer la contrainte `stretch` [71], puis la contrainte `pattern` [23].

4.1.1 Contrainte `stretch`

La contrainte `stretch` introduite en 2001 par Pesant [71], propose de restreindre le nombre d'apparitions consécutives d'une valeur dans une séquence de variables $X = x_0, x_2, \ldots, x_{n-1}$.

Définition 18 (Stretch) *Une sous-séquence $x_i, x_{i+1 \bmod n}, \ldots, x_{i+p \bmod n}$ est appelée* **stretch**, *ou* **succession** *en français, d'***étendue** $p+1$ *lorsque $x_i = x_{i+1 \bmod n} = \ldots = x_{i+p \bmod n}$ mais $x_{i-1 \bmod n} \neq x_i$ et $x_{i+p \bmod n} \neq x_{i+p+1 \bmod n}$.*

Définition 19 (Pattern) *Dans le cadre de la contrainte `stretch`, un* **pattern** *désigne deux successions d'activités de types différents (e.g. A,A,A,B,B sera représenté par le pattern AB).*

Définition 20 (Contrainte `stretch`) *Soient $X = x_0, x_2, \ldots, x_{n-1}$ une séquence de variables, D un ensemble de m valeurs prises dans les domaines de X, $\underline{\lambda}$ et $\overline{\lambda}$ deux vecteurs d'entiers de taille m, et P un ensemble de patterns. Pour tout x_i on notera $span_{x_i}$ l'étendue du stretch contenant x_i. La contrainte `stretch`$(X, \underline{\lambda}, \overline{\lambda}, P)$ assure que*

$$\forall i \in [\![0, n-1]\!], span_{x_i} \in [\![\underline{\lambda}_{x_i}, \overline{\lambda}_{x_i}]\!]$$

sous la condition que tout pattern de X appartiennent à P.

Dans son article, Pesant précise que les patterns autorisés P sont utilisés pour améliorer le filtrage, mais qu'ils ne sont pas assurés par la contrainte. Afin de répondre à ce problème, la contrainte `pattern` a été développée. Par ailleurs, Pesant propose une version de `stretch` basée sur la contrainte `regular` [73]

4.1.2 Contrainte `pattern`

Définition 21 (k-pattern) *On appellera un* **k-pattern** *une séquence de k éléments telle que deux éléments successifs ne prennent pas la même valeur.*

La contrainte `pattern` est décrite dans l'article de Bourdais et al. [23] comme l'une des primitives du système HIBISCUS pour la planification de personnel hospitalier. Elle assure que tous les k-patterns d'une séquence appartiennent à un ensemble donné de k-patterns autorisés.

Définition 22 (Contrainte `pattern`) *Soient $X = x_1, x_2, \ldots, x_n$ une séquence de variables prenant ses valeurs dans $D = d_1, d_2, \ldots, d_m$, et $D_e = d_{i1}, d_{i2}, \ldots, d_{il}$ la suite des valeurs prises par les stretchs successifs de X (e.g. si $X = A, A, B, B, B, A, C, C$, la séquence D_e vaut A, B, A, C).*
Soit $\mathcal{P}$ un ensemble de k-patterns. La contrainte `pattern`(X, P) est satisfaite, si et seulement si, toutes les sous-séquences de k éléments dans D_e sont dans $\mathcal{P}$.

4.2 Contrainte automate

Dans la continuité de ces travaux, les contraintes automates ont été développées pour prendre en compte des motifs de séquencement, autorisés ou interdits, plus complexes.

Comme nous l'avons vu dans le chapitre 3, les automates permettent de représenter des ensembles de mots, un mot étant composé d'une succession de symboles. En programmation par contraintes, la définition d'une contrainte, en intention ou en extension permet de définir un ensemble de solutions, chaque solution étant un ensemble de valeurs prises par les variables, satisfaisant la contrainte.

Par exemple, la contrainte $x_1 \neq x_2$ représente toutes les paires de valeurs prises dans $d_1 \times d_2$ tel que $x_1 \neq x_2$.

Ainsi, si l'on considère les symboles d'un alphabet comme les valeurs du domaine de variables et si l'on ordonne les variables, alors un mot sur l'alphabet correspond à une instantiation des variables. En conséquence, un automate sur l'alphabet définit un ensemble d'instantiations acceptables, de manière compacte.

4.2.1 Définition

Définition 23 (Contrainte automate) *Soient une séquence de variables $X = (x_1, \ldots, x_n)$ de domaines $d_1, \ldots, d_n$ et un automate $\mathcal{A}$. Une contrainte automate sur $\mathcal{A}$ assure que tout mot formé par la séquence de valeurs $(v_1, \ldots, v_n) \in (d_1 \times \ldots \times d_n)$ est accepté par $\mathcal{A}$.*

Cette contrainte a été introduite par Pesant [73] sous le nom de `regular`$(X, \mathcal{A})$, étant donné un automate fini déterministe $\mathcal{A} = (Q, \Sigma, \Delta, \{s\}, A)$. Par définition, les solutions de `regular`$(X, \mathcal{A})$ correspondent une à une aux chemins de n arcs connectant s à un sommet dans A dans le multigraphe orienté $\mathcal{A}$. Soit $\delta_i \in \Delta$ l'ensemble des transitions qui correspondent au i-ème arc d'un tel chemin, alors une valeur pour x_i est cohérente si et seulement si δ_i contient une transition étiquetée par cette valeur.

4.2.2 Algorithmes de filtrage

Incidemment, Pesant [73] et Beldiceanu et al [14] ont introduit deux approches orthogonales pour assurer la Cohérence d'Arc Généralisée (CAG) de la contrainte `regular` (voir la figure 4.1). L'approche proposée par Pesant [73] est de développer $\mathcal{A}$ comme un AFD acyclique $\mathcal{A}_n$ qui accepte uniquement les mots de longueur n. Par construction, $\mathcal{A}_n$ est un multigraphe à $n+1$ couches avec l'état s dans la couche 0 (la source), les états finals A dans la couche n (les puits), et où l'ensemble des arcs d'une couche quelconque i coïncide avec δ_i. Une première recherche étendue permet de maintenir la cohérence entre $\mathcal{A}_n$ et les domaines des variables en retirant les arcs dans δ_i dont les étiquettes ne sont pas dans le domaine de x_i, puis en retirant les nœuds et arcs qui ne sont connectés ni à une source ni à un puits.

Dans Beldiceanu et al [14], une contrainte `regular` est décomposée en n contraintes ternaires définies en extension et modélisant les ensembles $\delta_1, \delta_2, \ldots, \delta_n$. La décomposition introduit des variables d'état $q_0 \in \{s\}$, $q_1, \ldots, q_{n-1} \in Q$, $q_n \in A$ et des contraintes ternaires de transition $(q_{i-1}, x_i, q_i) \in \delta_i$. Le réseau des contraintes de transition étant Berge-acyclique, appliquer une consistance d'arc sur la décomposition permet d'atteindre la CAG sur `regular`. Soit l'AFD décrit ci-dessus appliqué à $X \in \{a,b\}\times\{a\}\times\{a,b\}\times\{b\}$. L'automate déployé de `regular` est présenté à gauche et le modèle décomposé sur la droite du schéma. Les transitions en pointillé sont filtrées dans les deux modèles.

Dans la première approche, un algorithme spécifique est défini pour maintenir l'ensemble des chemins supports, tandis que la seconde approche permet de laisser le solveur propager les contraintes de transition. Ces deux approches de filtrage sont orthogonales mais, selon l'ordre de propagation du second modèle, elles peuvent procéder de manière rigoureusement identique.

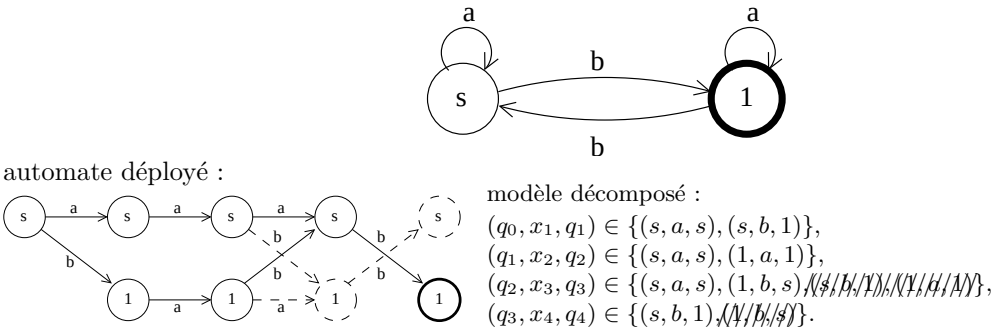


FIGURE 4.1 – Automate déployé et modèle décomposé pour l'auto interdisant un nombre pair de b

Assumons, sans perte de généralité, que Σ est l'union de domaines variables, alors l'exécution initiale de l'algorithme de Pesant pour la construction de $\mathcal{A}_n$ est réalisée en $O(n|\Delta|)$ en temps et en espace (avec $\Delta \leq |Q||\Sigma|$ si $\mathcal{A}$ est un AFD). Le filtrage incrémental procède par un parcours avant/arrière de $\mathcal{A}_n$ et possède cette même complexité de pire cas. En réalité, la complexité de l'algorithme dépend plus de la taille $\Delta_n|$ de l'automate déployé $\mathcal{A}_n$ que de la taille $|\Delta|$ de l'automate spécifié $\mathcal{A}$. Notons par exemple que lorsque l'automate spécifié $\mathcal{A}$ accepte uniquement des mots de taille n alors il est déjà déployé ($\mathcal{A} = \mathcal{A}_n$) et la première exécution de l'algorithme est en $O(|\Delta|)$. En pratique, de même que dans nos tests, $\mathcal{A}_n$ peut même être nettement plus petit que $\mathcal{A}$. Cela signifie que de nombreux états finals dans $\mathcal{A}$ ne peuvent pas être atteints en exactement n transitions. Le filtrage s'effectue alors en $O(|\Delta_n|)$ avec ici $|\Delta_n| \ll n|\Delta|$.

4.3 Extension des contraintes automates

L'introduction des contraintes sur les langages rationnels a ouvert la voie à deux extensions notables. La première extension considère des automates à transitions pondérées afin d'associer un coût à chaque séquence acceptée. La seconde extension considère des langages de plus haut niveau tels que définis par des grammaires hors-contexte (voir e.g. [56]).

4.3.1 Automates pondérés

La première extension de la contrainte `regular` fait directement suite aux travaux de Pesant. Demassey et al. [43] ont développé la contrainte `cost-regular` portant sur un automate associé à une matrice de coûts d'affectation, définissant un coût à chaque transition de l'automate. Dans le chapitre 3, nous avons introduit la notion d'automate pondéré. La contrainte `cost-regular` est définie sur une séquence de variables, une variable de coût, et un automate pondéré sur le semi-anneau $(\mathbb{R}, min, +)$.

Définition 24 (`cost-regular`) *Soit $\mathcal{A} = (Q, \Sigma, E, I, T)$ un automate pondéré dans le semi-anneau $(\mathbb{R}, \min, +)$. Soit X une séquence de variables entières et z une variable de borne à valeurs dans $\mathbb{R}$. La contrainte `cost-regular`$(z, X, \mathcal{A})$ est satisfaite, si et seulement si le mot m formé par les valeurs prises par les variables de X, est accepté par l'automate et si le poids $w(m)$ de m est égal à z.*

Le filtrage de la contrainte `cost-regular` est basé sur le filtrage par chemin de `regular` ainsi que sur la notion de plus petit et plus grand chemin dans l'automate déployé $\mathcal{A}_n$. Une dernière couche $n+1$ est ajoutée à $\mathcal{A}_n$ avec un seul état puits t. Pour tout état q de la couche n, un arc $(q, \sigma, t, 0)$ est ajouté. Soit s l'état initial tel que $I = \{s\}$. Soit $pcc(q, q') = min_{\pi \in \mathcal{A}(q,q')} w(\pi)$ le plus court chemin entre les états q et q'. Soit $pgc(q, q') = max_{\pi \in \mathcal{A}(q,q')} w(\pi)$ le plus grand chemin entre les états q et q'.

Un arc (q, σ, q', c) dans ce graphe est alors valide (il appartient à un chemin solution de la contrainte) si et seulement si :

$$pcc(s, q) + c + pcc(q', t) \leq \overline{z}$$

et

$$pgc(s, q) + c + pgc(q', t) \geq \underline{z}.$$

4.3.2 Relaxation de `regular`

Parallèlement à ces travaux, van Hoeve et al. [94] proposent de relâcher la contrainte `regular` en considérant les variations d'un mot accepté par un automate. Ils définissent la contrainte `soft-regular` de la manière suivante :

Définition 25 (`soft-regular`) *Soit $\mathcal{A} = (Q, \Sigma, \delta, \{s\}, T)$ un AFD et X une séquence de variables entières. Soit z une variable entière à valeurs dans $\mathbb{N}$ qui représente le coût de violation de la contrainte. Soit $d : \Sigma^* \times \Sigma^* \to \mathbb{N}$ une distance entre deux mots.*
La contrainte `soft-regular`$(z, X, \mathcal{A}, d)$ est satisfaite si et seulement si :

$$\min_{\sigma \in \mathcal{L}(\mathcal{A})} \{d(X, \sigma)\} \geq z.$$

Dans l'article, deux distances entre les mots sont considérées : la distance de Hamming et la distance d'édition. Ces deux distances permettent de mesurer, respectivement, le nombre de symboles distincts entre deux mots, et le nombre minimal d'insertions, de suppressions et de substitutions nécessaires pour obtenir un mot à partir d'un autre.

Le filtrage de la contrainte `soft-regular` est basé sur l'ajout d'arcs sur le graphe $\mathcal{A}_n$. Ces arcs ajoutés représentent les valeurs qui ne correspondent pas au langage de l'automate et ajoutent la possibilité de réaliser des insertions en autorisant les transitions entre les états d'une même couche. Les suppressions sont quant à elles gérées comme des ϵ-transitions. En distribuant les coûts sur ces nouveaux arcs, van Hoeve et al. montrent que l'algorithme de filtrage de `cost-regular` atteint la cohérence d'arc généralisée sur X par rapport à la borne supérieure de z.

Récemment, Métivier et al. [65] ont proposé d'utiliser la contrainte `cost-regular` pour modéliser la violation de l'utilisation de séquences interdites par l'automate en rajoutant explicitement ces séquences dans l'automate et en affectant un coût à la dernière transition de cette séquence. Dans leur article, les auteurs proposent également une variante de la contrainte `regular` (resp. `cost-regular`) : `regularCount` (resp. `cost-regularCount`) qui permet de combiner une contrainte `regular` (reps. `cost-regular`) avec une contrainte `atleast/atmost`.

4.3.3 Grammaires hors contexte

Au vu de l'apport en terme de modélisation et de résolution des contraintes sur les langages rationnels, une généralisation naturelle est l'utilisation de grammaire de type 2 dans la hiérarchie de Chomsky. Sellmann [86] a montré qu'il n'était pas possible au-delà du type 2 de proposer un propagateur permettant de faire la CAG en temps polynomial. En effet le problème du mot, c'est-à-dire décider de l'appartenance d'un mot donné à un langage donné, est décidable mais PSPACE-complet pour les grammaires de type 1 dites contextuelles.

4.4 Travaux connexes

4.4.1 Contraintes among et sequence

Une règle de séquencement particulière, limitant le nombre d'occurrences de certaines valeurs, est fréquente dans de nombreux problèmes, de planification et d'ordonnancement notamment. Elle se traduit au moyen de la contrainte among ou bien, de manière glissante, par la contrainte sequence.

La contrainte among est apparue sous le nom de between dans CHIP [13]. L'algorithme de Bessière et al. [18] établit la cohérence d'arc généralisée.

Définition 26 (Contrainte among) *Soient $X = (x_1, x_2, \ldots, x_n)$ une séquence de variables, S un ensemble de valeurs prises dans les domaines de X. Soient min et max deux constantes telles que $0 \leq min \leq max \leq n$.*

$$among(X, S, min, max) = \{(d_1, \ldots, d_n) \mid \forall i \in [\![1, n]\!]\ d_i \in D(x_i)\ et\ min \leq |\{i \in [\![1, n]\!] \mid d_i \in S\}| \leq max\}$$

La contrainte sequence, introduite par Beldiceanu et Contejean [13] peut être vue comme un ensemble de contraintes among imbriquées de manière glissante. Elle permet d'imposer la contrainte among sur chaque sous-séquence de variables d'une longueur donnée.

Définition 27 (Contrainte sequence) *Soient $X = (x_1, x_2, \ldots, x_n)$ une séquence de variables et S un ensemble de valeurs prises dans les domaines de X. Soient $q \in [\![1, n]\!]$, min et max deux constantes telles que $0 \leq min \leq max \leq n$.*

$$sequence(X, S, q, min, max) = \bigwedge_{i=1}^{n-q+1} among(X_{iq}, S, min, max), avec\ X_{iq} = x_i, \ldots, x_{i+q-1}.$$

Régin et Puget [78], ainsi que Beldiceanu et Carlsson [10], ont proposé un algorithme pour le filtrage de cette contrainte. Aucun de ces algorithmes ne réalise la cohérence d'arc généralisée. Plus récemment, van Hoeve et al. [93] ont introduit un algorithme polynomial ($O(n^3)$) atteignant la cohérence d'arc généralisée.

Du point de vue de la modélisation, la contrainte sequence est utilisée notamment dans le cadre du *car sequencing* où l'on cherche à planifier la production de voitures possédant différentes options sous des contraintes de capacité du type : « m voitures peuvent avoir l'option o toutes les n voitures » avec $m \leq n$. Pour la planification de personnel, elle permet de modéliser des règles courantes limitant l'apparition d'activités ou de groupes d'activités par période glissante. On peut par exemple imposer la règle « au maximum 2 quarts de nuit par période glissante de 7 jours » en posant la contrainte sequence$(X, \{N\}, 7, 0, 2)$.

Dans leur article, van Hoeve et al. [93] proposent une reformulation de la contrainte sequence en un automate fini déterministe. Cette reformulation permet d'atteindre la cohérence d'arc mais peut conduire à un automate de grande taille. La taille de l'automate dans le pire des cas est de l'ordre de $O(n2^q)$. Les auteurs précisent que dans la plupart des cas pratiques $q << n$ ce qui rend la reformulation exploitable.

Suite à ces travaux, Brand et al. [26] ont proposé un algorithme de complexité $O(n^2 log(n))$ réalisant la cohérence d'arc généralisée en encodant la contrainte sequence sous forme d'un problème en variables binaires et en décomposant le problème sous-jacent en de multiples inégalités.

Maher et al. [62] ont, quant à eux, proposé un algorithme de cohérence d'arc généralisée pour la contrainte sequence en $O(n^2)$, améliorant la performance de Brand et al. d'un facteur $O(log(n))$. Le filtrage repose sur l'utilisation d'algorithmes de flots. La différence majeure avec d'autres contraintes globales basées sur les flots est dans la représentation des valeurs des domaines comme un coût sur les arcs plutôt qu'une capacité.

4.4.2 Contrainte `slide`

Au-delà de `sequence`, la contrainte `regular` permet de modéliser n'importe quelle contrainte glissante. Récemment, Bessière et al. [21] ont introduit la meta-contrainte `slide`. Dans sa forme la plus générale, `slide` prend comme arguments une matrice de variables Y de taille $n \times p$ et une contrainte C d'arité pk avec $k \leq n$. `slide`(Y, C) est satisfaite si et seulement si $C(y^1_{i+1}, \ldots, y^p_{i+1}, \ldots, y^1_{i+k}, \ldots, y^p_{i+k})$ est satisfaite pour tout $0 \leq i \leq n - k$.

En utilisant la décomposition proposée en référence [14], `regular`$(X, \mathcal{A})$ peut être reformulée comme `slide`$([Q, X], C_\Delta)$, avec Q la séquence de variables d'état et C_Δ la contrainte de transition $C_\Delta(q, x, q', x') \equiv (q, x, q') \in \Delta$. Inversement [21], une contrainte `slide` peut être reformulée comme une contrainte `regular` mais cela nécessite d'énumérer tous les n-uplets valides de C. Cette reformulation peut cependant s'avérer utile dans le cadre de la planification (notamment en *car sequencing*) pour modéliser une contrainte de cardinalité glissante : la contrainte `sequence`.

4.4.3 Linéarisation de la contrainte

Comme le suggèrent les sections précédentes, de nombreux travaux ont été effectués autour des automates en programmation par contraintes. La contrainte `regular` a également inspiré des travaux dans la discipline connexe de la programmation linéaire en nombres entiers.

Dans leurs travaux, Côté et al. [40] proposent une transformation de la contrainte `regular` en un programme linéaire en nombres entiers. Rappelons que la condition nécessaire et suffisante d'existence d'une solution dans un problème modélisé à l'aide de la contrainte `regular` est l'existence d'un chemin dans le graphe déployé.

Côté et al. reprennent ce même graphe et montrent que la contrainte `regular` est satisfaite, si et seulement si une unité de flot peut circuler entre l'état initial du graphe et l'un de ses états finals.

L'extension de leur modèle pour une contrainte de type `cost-regular` est alors triviale puisqu'il s'agit d'ajouter de simples équations linéaires.

4.5 Applications

Les contraintes que nous avons présentées dans ce chapitre trouvent leurs applications dans de nombreux domaines. Trois en particulier font un usage intensif de ce type de contraintes :

- la reformulation automatique de contraintes globales ;
- la propagation de contraintes en extension ;
- la modélisation de règles de séquencement.

4.5.1 Reformulation automatique de contraintes globales

Les contraintes globales offrent de nombreux avantages quant à la modélisation de problèmes. Cependant le développement des algorithmes de filtrage et leur intégration au sein d'un solveur de contraintes représentent un effort non négligeable. Beldiceanu et al. [14] ont proposé de définir pour un grand nombre de contraintes l'automate permettant de vérifier si une solution satisfait la contrainte. En utilisant ces automates, ils proposent de dériver un algorithme de filtrage basé sur la décomposition présentée section 4.2. La liste des contraintes pouvant être modélisées à l'aide d'un automate se trouve dans le catalogue de contraintes globales [12].

4.5.2 Propagation de contraintes en extension

Dans leur article Richaud et al. [81] proposent d'utiliser un automate pour encoder un ensemble de n-uplets interdits appelés « nogood ». L'idée est de construire l'automate acceptant tous ces n-uplets et d'utiliser l'opération de complémentaire pour obtenir l'automate interdisant cet ensemble. L'algorithme de la contrainte `regular` est alors utilisé pour propager les propriétés apportées par ces n-uplets interdits.

De la même manière, une contrainte en extension est définie par un ensemble de n-uplets autorisés. La contrainte `regular` peut alors être utilisée avec l'automate représentant le langage acceptant tous les n-uplets de la contrainte. Les résultats présentés par Richaud et al. sont très encourageants car l'utilisation de `regular` permet d'obtenir la cohérence d'arc généralisée. Concernant l'enregistrement de « nogood », les résultats nuancés sont dus à la modification dynamique de l'automate ; les « nogood » étant ajoutés au fur et à mesure de la résolution.

4.5.3 Modélisation des règles de séquencement

Lors de la résolution de problèmes de planification de personnel, des règles métiers, des contraintes sur l'horaire ou la fréquence par exemple sont utilisées. Ces notions définissent ce que nous appelons des règles de séquencement. Au sein d'une séquence d'activités données, une règle de séquencement peut donner une information sur la façon de placer ces activités dans la séquence. On pourra par exemple établir une règle imposant un ordre d'apparition des activités (e.g. l'activité A doit apparaitre avant l'activité B) ou bien des règles d'interdiction ou d'obligation d'apparition d'une ou plusieurs activités (e.g. A et B ne doivent pas apparaître). Enfin, des contraintes temporelles peuvent être ajoutées à une règle de séquencement : une règle donnée ne sera valable par exemple qu'à partir du temps t ou entre les périodes p_1 et p_2.

La modélisation des règles de séquencement sera présentée plus en détail à travers une hiérarchisation et de nombreux exemples dans la seconde partie de cette thèse.

4.6 Conclusion

Dans ce chapitre, nous avons présenté les applications possibles des automates en programmation par contraintes pour la résolution de problèmes complexes. Cette présentation s'est appuyée sur les contraintes automates développées et utilisées notamment dans les travaux de Gilles Pesant et Nicolas Beldiceanu.

Lorsque l'on conçoit un logiciel d'aide à la décision, la modélisation du problème que l'on cherche à résoudre est une étape critique et difficile. Les outils que nous avons présentés dans la première partie de cette thèse, les automates et les expressions rationnelles, vont nous permettre de répondre au problème de la modélisation en deuxième partie de cette thèse. Nous verrons également dans la dernière partie de cette thèse qu'ils sont un support important pour la résolution des problèmes de planification de personnel.

Deuxième partie

De la modélisation au filtrage de règles de séquencement

Chapitre 5

Modélisation systématique de règles de séquencement

Sommaire

5.1 Introduction

Dans ce chapitre, nous nous intéressons à la modélisation mathématique de règles de séquencement.

La problématique que nous souhaitons résoudre est simple : comment modéliser, de manière compacte et exploitable, des règles de séquencement complexes, apparaissant dans divers problèmes telle la planification de personnel ?

Notre approche se base sur la théorie des langages. En considérant qu'un élément d'une séquence est un symbole, alors cette séquence forme un mot dans un langage qu'il reste à déterminer. Sans règle de séquencement, une séquence d'activités prise dans un ensemble fini peut se voir comme l'ensemble des mots générés par l'alphabet des activités (i.e. toutes les activités possibles).

Plus formellement, soit Σ, l'ensemble des activités. Alors l'ensemble des séquences d'activités est représenté par le langage Σ^*. Une règle de séquencement viendra restreindre cet ensemble de manière à décrire uniquement l'ensemble des séquences autorisées.

Considérons maintenant l'ensemble des séquences admissibles pour des règles de séquencement données. Cet ensemble peut être vu comme un ensemble de mots formant un langage. La problématique

revient alors à représenter, de manière compacte, ce langage. Or, comme nous l'avons vu au chapitre 3, certaines classes de langages peuvent être représentées par différents types d'automates (i.e. automates simples, automates pondérés).

Ainsi, nous proposons dans ce chapitre de détailler pour un grand nombre de règles de séquencement, la modélisation sous forme d'équations simples ou d'automates. De plus, afin d'automatiser cette modélisation, nous proposons également les algorithmes permettant de construire de manière systématique les automates. Les nombreux exemples illustrant les règles de séquencement sont issus pour la plupart des problèmes de planification de personnel qui seront abordés dans un prochain chapitre. Il est à noter néanmoins que l'utilisation d'automates pour modéliser des règles de séquencement peut être appliquée dans bien d'autres cadres.

Dans un premier temps nous nous intéresserons aux règles de séquencement imposées à une séquence, puis dans un second temps nous décrirons les procédés utilisés pour modéliser des règles de séquencement souples.

5.2 Règles de séquencement obligatoires

Définition 28 (Règle de séquencement) *Nous appellerons règle de séquencement toute information, règle ou contrainte qui modifie les activités autorisées pour une séquence donnée. Une séquence d'activités $a \in A$ de longueur n sera représentée par une suite de variables $S_1, S_2, \ldots, S_n$ qui prennent leurs valeurs dans A.*

Au cours de nos travaux, nous avons rencontré différents types de règles. Afin de générer automatiquement les automates, compteurs ou coupes dans les domaines nécessaires à la modélisation complète d'un problème formulé sous formes de contraintes, nous avons construit trois catégories de règles portant sur :

- les activités ;
- les stretch d'activités ;
- les motifs d'activités.

De cette hiérarchie, nous pouvons extraire trois catégories : la règle est-elle interdite, est-elle obligatoire ou son nombre d'occurrences est-il limité ? Dans le premier cas, il s'agira par exemple de forcer une activité donnée à une date précise. Pour un motif d'activités d'occurrence limitée, il s'agira de vérifier que le nombre d'apparitions d'un motif défini par une expression rationnelle apparaît au moins et au plus un nombre donné de fois.

Chacune de ces règles peut être applicable à date fixe, ou n'importe quand au cours de la séquence que l'on cherche à planifier. Pour chacune de ces règles, nous proposons un objet mathématique pour la représenter : une égalité ou une inégalité, une expression rationnelle (ou son automate équivalent) ou un automate pondéré.

Le tableau suivant résume les différentes combinaisons de règles que nous traitons dans cette thèse ainsi que la modélisation sous forme d'équations simples ($=$ pour égalité, $\neq$ pour inégalité) ou sous forme d'expression rationnelle (`er`) et/ou d'un compteur (`cpt`) lorsqu'il s'agit d'un automate pondéré.

5.2.1 Activités

Les règles qui interdisent, ou forcent une activité à être effectuée, sont très simples à modéliser. Dans le cas d'une date fixe, il suffit de modifier la variable correspondant à la période et l'employé concerné à

	date fixe		période		
	obligatoire	interdit	obligatoire	interdit	occ. limite
activité	$=$	$\neq$	`er` ou `cpt`	`cpt` ou $\neq$	`cpt`
stretch	`er`	`er`	`er`	`er`	`er` + `cpt`
motif	`er`	`er`	`er`	`er`	`er` + `cpt`

TABLE 5.1 – Tableau récapitulatif de la hiérarchie des règles de séquencement obligatoires

l'aide d'une contrainte unaire d'égalité (=). Ainsi, pour forcer l'activité a au temps t dans une séquence, on pose :

$$S_t = a.$$

De manière symétrique, pour interdire cette même activité en même temps, on posera l'inégalité :

$$S_t^e \neq a.$$

Ces deux équations auront pour effet de modifier les domaines des variables concernées.

Notre premier exemple portera sur une règle apparaissant dans la grande distribution où un salarié doit être présent obligatoirement le samedi et jamais le dimanche. Soit un employé e dont on doit planifier un horaire pour une période de sept jours représentant une semaine : ses activités possibles sont R pour repos, M pour matin et S pour soir. Un repos est imposé le dimanche, mais interdit le samedi. Cette règle est présente dans les grands magasins ouverts le samedi, jour où tous les salariés doivent être présents et fermés le dimanche. On pose alors :

$$S_5^e \neq R$$

et

$$S_6^e = R.$$

Les domaines deviennent respectivement $\{M, S\}$ et $\{R\}$.

Les règles concernant une activité deviennent plus complexes lorsqu'elles agissent non plus à une date fixe mais sur une période (i.e. une suite de dates fixes). Une activité a obligatoire sur la période de temps $T = [\![t_b, t_e]\!]$ peut se traduire par la règle « au moins un a sur cette période ». Si un automate peut remplir cette fonction, en forçant tout chemin valide dans l'automate à passer par une transition a, cette solution n'est pas satisfaisante pour plusieurs raisons. Tout d'abord, la période concernée, comprise entre t_b et t_e ne correspond pas forcément à toute la période. Construire un automate sur une sous-période est possible (nous utiliserons d'ailleurs cette technique pour d'autres types de règles) mais complexe par rapport à la règle demandée. Ensuite, si l'on considère un automate représentant cette unique règle sur toute une période de sept jours, l'automate construit aura au minimum le double d'états de par le caractère flottant de la règle. L'activité a pouvant apparaître n'importe quand durant la période, il faut, à chaque état où une transition a est possible, dupliquer la fin de l'automate pour différencier le cas où l'activité a a été effectuée ou non.

Sur une période de sept jours avec cette unique règle, l'automate ne possède que deux états, mais associé à d'autres règles cela complexifie l'automate de manière inutile. L'exemple suivant 5.1 reprend les mêmes activités M, S et R.

L'expression rationnelle correspondante à partir de laquelle nous avons construit l'automate signifie que l'on peut faire n'importe quelle activité autre qu'un repos, mais qu'un repos est nécessaire pour être une chaîne valide. Il faut lire : matin ou soir zéro fois ou plus, suivi d'un repos, suivi de n'importe quelle

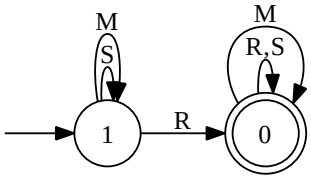


FIGURE 5.1 – Automate représentant la règle « au moins un repos durant la période »

activité zéro fois ou plus :

$$(M|S) * R(M|S|R) * .$$

Modifions par exemple la règle de manière à ce qu'elle s'applique uniquement sur la période $T \in [\![3, 6]\!]$. Nous sommes alors obligés d'expliciter chaque transition de manière à ce qu'elle corresponde à une période donnée. L'expression rationnelle associée à notre règle devient alors :

$$(M|S|R)\{2\}(M|S) * R(M|S|R) * (M|S|R).$$

Il faut ici lire, n'importe quelle activité deux fois, puis la règle édictée précédemment, puis pour la dernière période n'importe quelle activité à nouveau. La taille de la séquence de variables étant 7, cela impose bien, sur la période $T \in [\![3, 6]\!]$, qu'au moins un repos soit effectué. L'automate correspondant est construit à partir de cette expression figure 5.2.

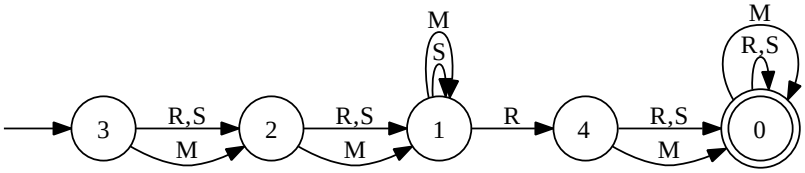


FIGURE 5.2 – Automate représentant la règle « au moins un repos durant la période $T \in [\![3, 6]\!]$ »

L'alternative à ces automates qui modélisent une règle somme toute assez simple est d'utiliser un compteur. Nous rajoutons alors à notre modèle une variable compteur y_r, où r est l'indice du compteur. Le domaine de cette variable dépend de la règle. Si l'on reprend notre exemple et que l'on souhaite qu'il y ait au moins un repos dans un horaire de 7 jours, il faut d'abord poser $y_r = [1, 7]$. Nous ajoutons aussi un coût d'affectation c_{tar} qui indique le coût d'affectation pour chaque temps t d'une activité a pour le compteur r. Il suffit alors de poser les coûts suivants :

$$\forall t \in [\![1, 7]\!],\ c_{tRr} = 1$$

et

$$\forall t \in [\![1, 7]\!], \forall a \in A/\{R\},\ c_{tar} = 0.$$

Ainsi, à chaque fois qu'une transition portant un repos est utilisée dans la semaine (période de 1 à 7), un coût de 1 est attribué au compteur r. Un mot sera accepté dans cet automate si et seulement si sa lecture dans l'automate termine sur un état final et le compteur est compris entre 1 et 7.

L'automate pondéré figure 5.3 représente la règle « au moins un repos sur la période $T \in [\![1, 7]\!]$ ». La dernière transition menant vers le domaine de la variable compteur indique la condition sur cette variable pour que le mot soit accepté par l'automate.

Remarque : cet automate n'est pas minimal car un seul état suffirait. Nous avons volontairement imposé la taille minimale de la séquence à 1 pour plus de lisibilité.

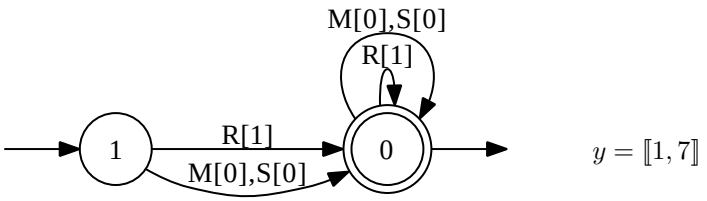


FIGURE 5.3 – Automate pondéré représentant la règle « au moins un repos durant la période $T \in [\![1, 7]\!]$ »

Si comme précédemment nous souhaitons imposer cette règle uniquement sur la période $T \in [\![3, 6]\!]$, il faudra alors expliciter les différents états, afin de ne pondérer les transitions correspondant à un repos que lorsqu'elles appartiennent à la période donnée. Afin de le faire, nous devons déployer l'automate sur la période de 7 jours. L'automate pondéré que nous obtenons est présenté figure 5.4.

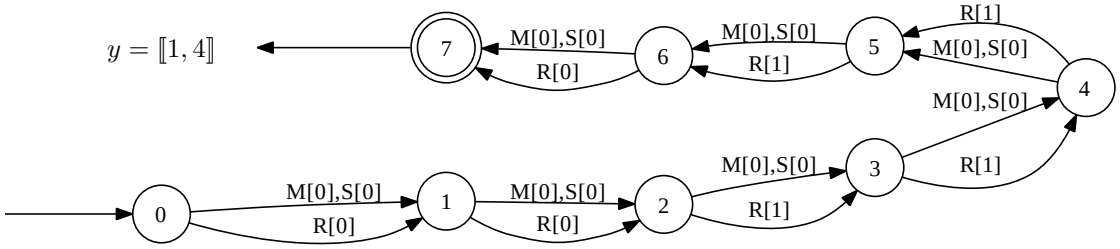


FIGURE 5.4 – Automate pondéré représentant la règle « au moins un repos durant la période $T = [\![3, 6]\!]$ »

Le cas où une activité est interdite est ici plus simple puisqu'il suffit de supprimer pour la période donnée les valeurs des domaines des variables S_t. Comme précédemment, l'usage d'une inégalité sur les variables concernées interdira l'utilisation de l'activité sur cette période. On pourrait également ajouter un compteur de repos sur la période et forcer la variable compteur y_r à zéro. Néanmoins, cela nécessite l'ajout d'une dimension de coût qui bien que transparente en terme de complexité provoque un surplus de calcul inutile ici.

La dernière contrainte que nous souhaitons pouvoir imposer sur une ou un ensemble d'activités, est son occurrence au sein d'une période donnée. Pour un horaire de deux semaines composé de nuits, de jours et de repos, nous pourrions par exemple souhaiter que la première semaine, seulement deux journées soient travaillées au maximum, tandis que la seconde semaine, cinq journées travaillées soient nécessaires.

La solution que nous proposons à ce type de règle est d'utiliser encore une fois des compteurs. Nous avons besoin d'un compteur par période contrainte. Comme précédemment, nous créons un premier compteur $y_1 = [0, 2]$ qui limitera les activités travaillées (nuit ou jour) à deux la première semaine. De la même façon, un deuxième compteur $y_2 = [5, 7]$ bornera le nombre d'activités travaillées la deuxième semaine.

Il reste alors à placer les coûts sur les transitions de l'automate pondéré. Notons qu'ici chaque transition portera un vecteur de coût et non plus un coût unique. Nous créons donc ici un automate multi-pondéré. Pour construire l'automate sans compteur nous utilisons l'expression rationnelle suivante qui explicite les 14 jours de la séquence que l'on souhaite construire :

$$(J|N|R)\{14\}.$$

La matrice des coûts d'affectation c_{tar} se remplit alors ainsi :

$$\forall t \in [\![1,7]\!], c_{tar} = \begin{cases} 1 \text{ si } a \in \{J,N\} \text{ et } r = 1 \\ 0 \text{ sinon} \end{cases}$$

$$\forall t \in [\![8,14]\!], c_{tar} = \begin{cases} 1 \text{ si } a \in \{J,N\} \text{ et } r = 2 \\ 0 \text{ sinon} \end{cases}$$

La transformation de cette expression rationnelle et de sa matrice de coûts donne l'automate multi-pondéré en figure 5.5.

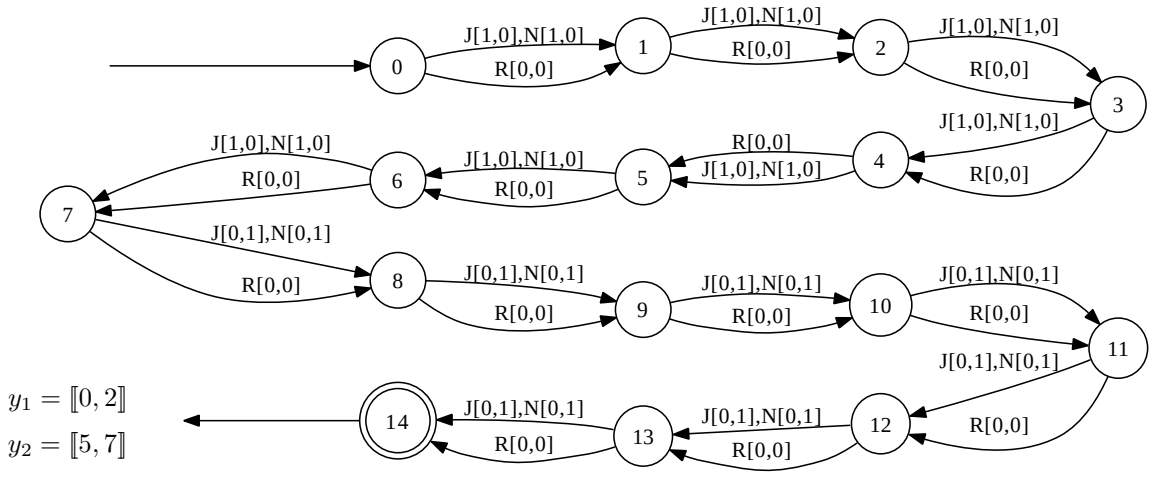


FIGURE 5.5 – Automate multi-pondéré représentant les règles « au plus deux jours travaillés la première semaine » et « au moins 5 jours travaillés la deuxième semaine »

Nous venons de faire une liste exhaustive des différents types de règles concernant les activités que nous avons rencontrées au cours de cette thèse. Si certaines règles, telles les obligations ou interdictions à date fixe, sont très simples à modéliser automatiquement à l'aide d'égalités ou d'inégalités, contrôler le nombre d'occurrences d'une activité est une problématique différente. L'intégrer dans un automate sans compteur nécessitera d'augmenter parfois considérablement la taille de l'automate. Les exemples que nous avons donnés n'illustrent pas complètement l'intérêt d'utiliser un compteur, puisque les automates ne portent aucune autre règle. Dans les cas réels, que nous présenterons plus tard, nous ajouterons des compteurs à des automates représentant des règles plus complexes, tels les motifs d'activités imposés. Or, l'intersection d'un automate, tel que celui présenté figure 5.2, de taille $n = 5$, avec un automate représentant un motif de taille n' peut produire un automate de taille $n' \times n$. Avec de nombreuses activités à compter et de nombreux motifs à imposer, la taille de l'automate deviendra très vite critique.

En conclusion, dès lors que compter, une ou un ensemble d'activités est nécessaire, l'utilisation d'un

compteur associé à un automate, représentant à minima une autre règle concernant un motif obligatoire, est préconisée. Plus l'automate comportera des règles complexes sur les motifs, plus l'utilisation d'un compteur sera pertinente.

Le tableau 5.2 récapitule la modélisation que nous utilisons pour modéliser automatiquement des règles concernant les activités d'un horaire à planifier.

	activité	modélisation retenue	Contraintes utilisées pour résolution
Date fixe	obligatoire	modification de domaine	$=$ ou $\neq$
	interdite	modification de domaine	$\neq$
Période	obligatoire	ajout d'un compteur	automate (multi-)pondéré
	interdite	modification de domaine	$\neq$
	à occurrence limitée	ajout d'un compteur	automate (multi-)pondéré

TABLE 5.2 – Tableau récapitulatif des modèles et contraintes pour les règles d'activités

5.2.2 Succession d'activités

Dans la section 5.2.1, nous avons présenté un ensemble de règles portant sur une ou un ensemble d'activités. Ici, nous nous intéressons à la modélisation de séquences d'activités de même type. Une activité $a \in A$ doit être suivie par un certain nombre de cette même activité. Cette règle se généralise à un ensemble d'activités de même type. La présence de ce type de règles est par exemple naturelle dans les problèmes de planification de personnel : il est en effet courant de demander à un employé de travailler au moins trois nuits consécutives ou au moins cinq jours consécutifs. Cette règle apparaît souvent dans les usines pratiquant les 3×8 par exemple. Comme pour la section sur les activités, cette règle peut être imposée à date fixe ou sur une période. De même, on peut envisager de la rendre obligatoire ou interdite sur un certain nombre, fixé à l'avance, de quarts. Finalement, nous verrons que ce nombre de quarts peut devenir variable. On cherchera alors à maintenir le nombre d'occurrences consécutives d'une activité.

Règle de consécutivité. « Une matinée le deuxième jour est suivie de deux autres matinées »

Cette règle impose que si l'on travaille le matin le deuxième jour de l'horaire, il faut alors travailler le matin encore deux jours de plus, c'est-à-dire le troisième et le quatrième jour de l'horaire. Nous modéliserons cette règle à l'aide d'un automate, explicitant les activités disponibles le deuxième jour. Si une matinée est effectuée le deuxième jour, la transition la représentant doit mener à une succession de matinées.

L'expression rationnelle suivante modélise cette règle. Nous utiliserons ici encore M pour désigner une matinée, S pour une soirée et R pour un repos :

$$(M|S|R)(MMM|((S|R)(M|S|R)(M|S|R)))(M|S|R)*.$$

Il faut lire ici, n'importe quelle activité, suivie de :

- soit trois matinées ;
- soit toute activité sauf une matinée puis n'importe quelle activité.

L'automate minimal équivalent est présenté figure 5.6. La différenciation entre la séquence matinée et les autres séquences le deuxième jour se fait à partir de l'état 1.

Le contraire d'obliger une succession d'activités d'une longueur donnée à une date fixe est d'interdire l'apparition d'une telle séquence. Le principe de construction de l'automate est de cette manière identique

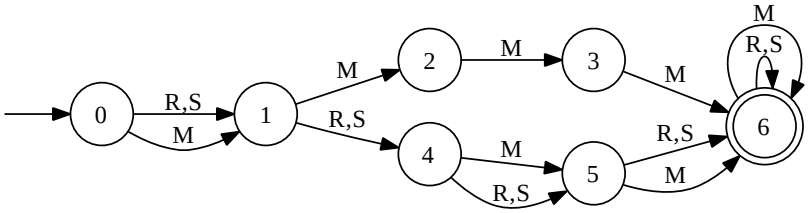


FIGURE 5.6 – Automate représentant la règle « une matinée le deuxième jour est suivie de deux autres matinées »

à l'obligation d'avoir une succession. Nous différencions donc les activités dont nous souhaitons contrôler la série. Prenons par exemple la règle « deux repos consécutifs au maximum le premier jour ». Dès le début de la séquence que l'on souhaite caractériser nous différencions le cas où deux repos apparaissent. Au sein d'une expression rationnelle, cela se traduit par deux repos R successifs RR. Après deux repos, une activité travaillée est donc obligatoire, la séquence devient donc $RR(M|S)$ pour deux repos suivis d'un matin ou d'un soir. Il faut maintenant ajouter les deux autres cas possibles, c'est-à-dire lorsque l'un des deux premiers jours n'est pas un repos. Dans ce cas, la règle ne s'applique pas. On écrit donc, un jour travaillé suivi de n'importe quelle activité $(JlN)(J|N|R)$ ou un repos suivi d'un jour travaillé $R(J|N)$. L'expression rationnelle décrivant la règle est donc :

$$((M|S)|(RR(M|S))|(R(M|S)))(M|S|R)*.$$

Figure 5.7, l'automate fini déterministe issu de cette expression rationnelle illustre les différents choix possibles dès l'état 0. Néanmoins, la construction d'un tel automate n'est pas satisfaisante. En effet, nous nous sommes contentés d'énumérer les différents cas possibles de manière logique à partir de la règle. Or, dans une optique de construction automatique cela n'est pas réalisable simplement. De plus, si l'exemple énoncé ne propose que trois alternatives, ce nombre peut augmenter rapidement sur des séquences plus grandes avec des activités en plus grand nombre.

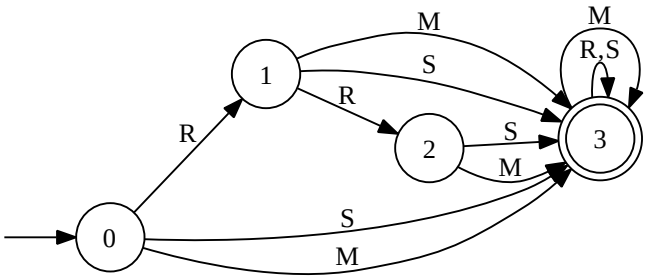


FIGURE 5.7 – Automate représentant la règle « deux repos consécutifs au maximum le premier jour » (méthode 1)

L'alternative de construction que nous proposons est donc d'utiliser la négation dans l'expression rationnelle. Cette dernière devient alors :

$$\sim (RRR + (M|S|R)*).$$

Cette expression se traduit par « il est interdit d'avoir trois repos, ou plus, suivis de n'importe quelle activité ». Une fois compilée, l'expression rationnelle produit l'automate figure 5.8. Comme ils sont construits automatiquement à partir des expressions rationnelles, les automates figures 5.7 et 5.8 n'ont pas la même forme. Ils ont cependant les mêmes états et les mêmes transitions : ils sont équivalents.

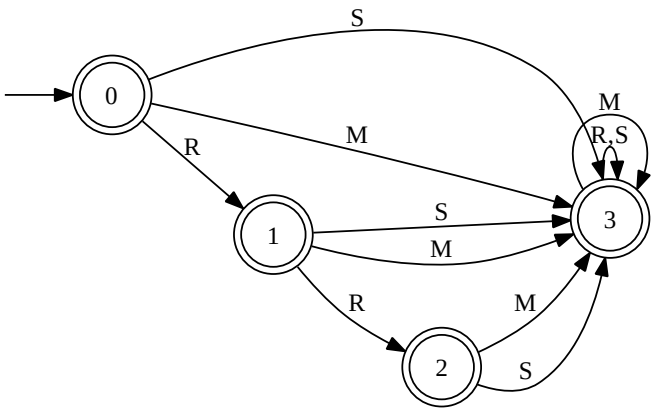


FIGURE 5.8 – Automate représentant la règle « deux repos consécutifs au maximum le premier jour » (méthode 2)

Toujours autour d'une date fixe, nous pouvons également imposer qu'une succession d'une même activité ait une taille comprise entre deux bornes fixes. Soit une activité $a \in A$. Imposer une succession de a de longueur $n \in [\![l, u]\!]$ se fait à l'aide d'un automate de la même manière que pour imposer ou limiter la longueur de la succession. La seule différence est qu'à la place d'un état attestant que la règle est vérifiée et qu'une autre activité est autorisée, il y a aura ici un ensemble d'états. Si un de ces états est atteint, cela signifie que l'on a effectué une succession de a de longueur $n \in [\![l, u]\!]$. Nous pouvons donc généraliser les deux cas précédents à l'aide de cette construction. L'automate figure 5.9 représente cette généralisation. Dans cet automate, toute transition a est suivie d'au moins $l - 1$ transition a afin de respecter la borne inférieure de longueur de la succession. Il existera ensuite $u - l$ autres transitions autorisant de reprendre une autre activité (états en rouge). La règle est, par souci de concision, imposée ici au jour 0.

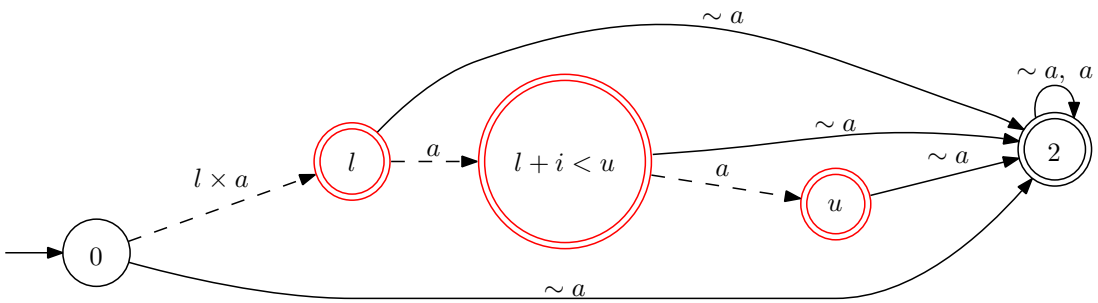


FIGURE 5.9 – Automate bornant la taille d'une succession d'activités au jour 0

Cet automate est simple à construire automatiquement à partir de la donnée des activités et de la longueur demandée pour la succession d'activités. Il en est tout autre pour l'expression rationnelle associée qui, quant à elle, se doit d'expliciter comment arriver à chaque état rouge $(l, (l + i < u)$ et $u)$.

C'est pourquoi, dans le cas de règles concernant la taille obligatoire de successions d'activités, nous créons directement l'automate, sans passer par l'expression rationnelle équivalente. Pour cela, nous avons écrit un algorithme automatisant la création d'un automate bornant la taille de successions d'activités. L'algorithme 1 explicite d'abord les transitions jusqu'à la période t où la règle s'applique puis différencie les cas :

- l'activité appartient à l'ensemble d'activités de la succession ;
- l'activité n'appartient pas à cet ensemble.

Dans le premier cas, il faut créer autant d'états que la valeur de la borne u. En reliant ces états avec des transitions a, on s'assure d'avoir une succession de a de taille maximum u. Tous les états accessibles depuis la période t en l transitions ou plus doivent être connectés à un état final par une transition $\sim a$ (tout sauf a). Dans le deuxième cas, la règle ne s'applique pas, on peut donc faire toute activité à partir de l'état suivant.

Entrées: Une activité a, des bornes l et u de la longueur de la succession d'activités, le jour t où la règle s'applique
Sorties: Un automate acceptant uniquement les successions d'activités de longueur $n \in [\![l, u]\!]$
pour tous les $i \in [\![0, t]\!]$ **faire**
  Créer un nouvel état q_i
fin
pour tous les $i \in [\![0, t-1]\!]$ **faire**
  Ajouter la transition $q_i \xrightarrow{\sim a} q_{i+1}$;
  Ajouter la transition $q_i \xrightarrow{a} q_{i+1}$;
fin
pour tous les $i \in [\![1, u]\!]$ **faire**
  Créer un nouvel état final q_{t+i};
  Ajouter la transition $q_{t+i-1} \xrightarrow{a} q_{t+i}$;
fin
Créer un nouvel état final q_e;
pour tous les $i \in [\![t+l, t+u]\!]$ **faire**
  Ajouter la transition $q_i \xrightarrow{\sim a} q_e$;
fin
Ajouter la transition $q_t \xrightarrow{\sim a} q_e$;

Algorithm 1: Création d'un automate pour les règles de successions

Nous pouvons illustrer ce cas général en imposant que, dans un ensemble d'activités jour, nuit et repos $\{J, N, R\}$, entre deux et quatre jours travaillés successivement soient obligatoires si l'on travaille le deuxième jour. La construction de l'automate se fait alors à l'aide de l'algorithme 1. Le résultat est donné figure 5.10.

Il est légitime de se demander comment transformer ces règles lorsqu'elles sont généralisées à toute la séquence. On pourrait naïvement décomposer à chaque état, les transitions, de manière à expliciter la règle à chaque fois. Si le principe est correct en soit, lorsque nous traiterons des règles plus complexes, dans des automates déjà construits, et donc plus lourds, cela implique qu'il faudra à chaque état, dupliquer l'automate.

Nous proposons donc de créer un automate représentant uniquement la règle, de manière à ce qu'un état particulier soit considéré comme entrant pour la règle. Ainsi, si la succession d'activités considérée

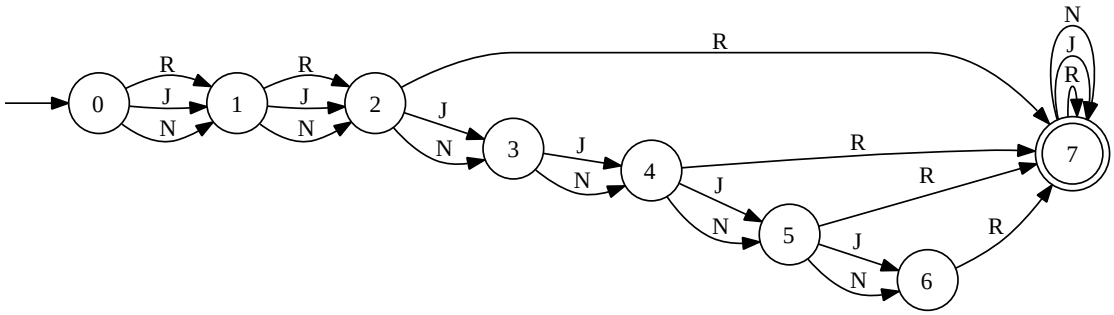


FIGURE 5.10 – Automate représentant la règle « deux et quatre jours travaillés successivement sont obligatoires si l'on travaille le deuxième jour. »

commence par l'activité $a \in A$, toutes les transitions marquées par une autre activité doivent arriver à un même état. De cet état la règle peut commencer à être lue si une activité a est utilisée.

L'automate figure 5.11 représente la forme générale d'une succession d'activités a ayant pour longueur $n \in [\![l, u]\!]$ sur toute la période.

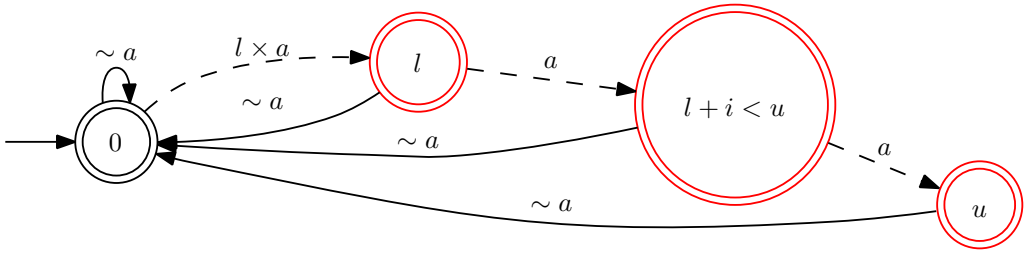


FIGURE 5.11 – Automate représentant la forme générale d'une règle de successions d'activités sur toute une période

La construction automatique de cet automate se fait à l'aide de l'algorithme 2 qui reprend le principe de celui décrit précédemment.

Nous pouvons maintenant automatiser l'obligation ou l'interdiction d'avoir des successions d'activités lors d'un horaire, que ce soit à une date fixe ou pendant une période déterminée.

Une règle qui peut apparaître également est de compter le nombre de ces successions pendant un horaire. Ce type de règle apparaît lorsque l'on planifie un employé sur de plus longues périodes, par exemple un mois. Au cours de ce mois, l'employeur peut souhaiter qu'une personne travaille uniquement par périodes consécutives. Ainsi dans certaines casernes, les pompiers doivent travailler deux ou trois jours consécutifs quatre fois par mois.

Pour compter le nombre de successions d'activités apparaissant dans un horaire nous utiliserons un automate pondéré afin d'une part, d'assurer la règle sur le nombre de jours consécutifs de l'activité choisie, et d'autre part, pour compter le nombre de fois où une telle succession apparaît.

Pour ce faire, nous transformons l'automate général figure 5.11 en un automate pondéré en y intégrant des coûts. Tout arc sortant de la séquence se voit attribuer un coût de 1, 0 pour tous les autres arcs. Ainsi, à chaque fois qu'un arc ayant pour coût 1 est emprunté, cela veut dire qu'une succession d'activités

Entrées: Une activité a, des bornes l et u de la longueur de la succession
Sorties: Un automate acceptant les successions d'activités de longueur $n \in [\![l, u]\!]$ au cours de l'horaire
pour tous les $i \in [\![0, u]\!]$ **faire**
  Créer un nouvel état final q_i
fin
pour tous les $i \in [\![0, u-1]\!]$ **faire**
  Ajouter la transition $q_i \xrightarrow{a} q_{i+1}$;
fin
pour tous les $i \in [\![l, u]\!]$ **faire**
  Ajouter la transition $q_i \xrightarrow{\sim a} q_0$;
fin
Ajouter la transition $q_0 \xrightarrow{\sim a} q_0$;

Algorithm 2: Création d'un automate pour les successions d'activités

a vient d'être effectuée. Le coût sur cet arc permet d'ajouter à une variable compteur y le passage par la succession. La variable de compteur $y = [\![s, S]\!]$ est ajoutée de manière à borner le nombre de successions entre s et S. L'automate figure 5.12 représente un automate pondéré imposant des successions d'activités de longueur $n \in [\![l, u]\!]$.

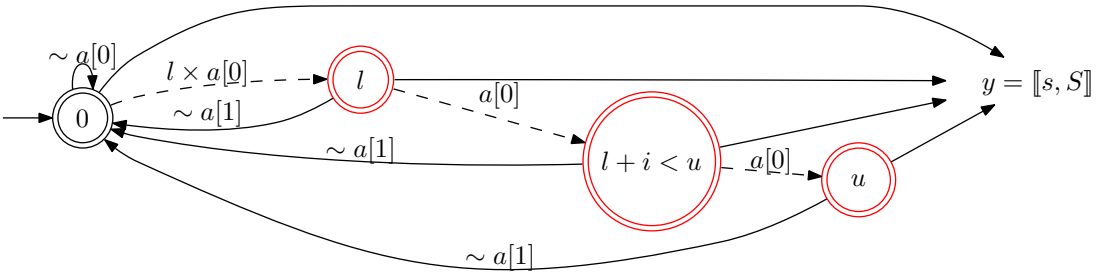


FIGURE 5.12 – Automate pondéré de successions d'activités

L'algorithme 3 utilisé pour créer ce type d'automate est peu différent de celui développé pour les automates d'obligation ou d'interdiction (algorithme 3 de successions d'activités). Il est en effet suffisant d'ajouter les coûts de 1 lors de la construction des transitions $\sim a$ vers l'état initial.

5.2.3 Motifs d'activités

Nous pouvons désormais générer automatiquement des expressions rationnelles ou des automates, pour représenter des règles d'interdiction, d'obligation ou de comptage portant sur des activités ou des successions d'activités. La généralisation de ces techniques, comme annoncé dans l'introduction de cette section consiste à faire la même chose pour des motifs d'activités. Nous entendons par motif d'activités toute succession d'activités ou d'ensemble d'activités qui peuvent être représentées par une expression rationnelle (ou son automate équivalent).

Cette généralisation est importante puisqu'elle permet de modéliser des règles plus complexes apparaissant dans les problèmes de planification de personnel. Si cette généralisation permet également de modéliser les règles concernant les activités et les successions d'activités, les modèles d'automates servant

Entrées: Une activité a, des bornes l et u de la longueur de la succession, les bornes s et S du nombre de successions
Sorties: Un automate acceptant les successions d'activités de longueur $n \in [\![l, u]\!]$ entre s et S fois, une variable compteur y
pour tous les $i \in [\![0, u]\!]$ **faire**
| Créer un nouvel état final q_i
fin
pour tous les $i \in [\![0, u-1]\!]$ **faire**
| Ajouter la transition $q_i \xrightarrow{a} q_{i+1}$ de coût 0;
fin
Créer une nouvelle variable $y = [\![s, S]\!]$;
pour tous les $i \in [\![l, u]\!]$ **faire**
| Ajouter la transition $q_i \xrightarrow{\sim a} q_0$ de coût 1;
fin
Ajouter la transition $q_0 \xrightarrow{\sim a} q_0$ de coût 0;

Algorithm 3: Création d'un automate pour compter les successions d'activités

à modéliser des règles sur les activités ou les successions d'activités ne sont pas à jeter pour autant. En effet, les processus de modélisations proposés, offriront lors de la résolution des modèles plus efficaces.

Ainsi, lorsque l'on différencie les règles sur les activités et sur les successions d'activités par exemple, cela permet d'utiliser un compteur pour savoir combien de fois une activité donnée apparaît. Si nous les modélisions avec un automate comptant les successions d'activités de taille 1, nous dupliquerions un grand nombre d'états. L'automate final représenterait donc les mêmes ensembles de solutions mais serait plus gros, donc plus lent à traiter par le solveur.

Forcer l'apparition d'un motif à date fixe. L'apparition d'un motif obligatoire est une règle qui peut apparaître dans le cadre de la production par exemple. En effet, en raison d'une commande ou d'une activité spéciale, un enchaînement de tâches pourrait devoir être effectué à une date précise. Forcer un motif peut paraître simple si l'on suppose que ce motif n'est constitué que de successions d'activités uniques et prédéterminées. Or, il arrive souvent que pour un même procédé, plusieurs méthodes soient possibles. Dans ce cas, le motif représentant la réalisation complète d'un procédé s'exprime de manière plus complexe. Ainsi, choisir la bonne succession d'activités pour optimiser l'horaire ne peut se faire a priori. Dans d'autres contextes plus souples, il se peut que l'on veuille laisser une certaine liberté dans le choix des activités avec des passages obligatoires.

La puissance des expressions rationnelles nous permet encore une fois de modéliser ce genre de contraintes. Supposons que nous voulions imposer qu'à la deuxième période de la journée, après réception de pièces à usiner, un ouvrier soit contraint d'effectuer ces différentes tâches : (D)écoupage et (P)einture suivies de (S)oudure puis (M)ontage. Enfin les pièces sont (E)ntreposées ou (T)riées ou e(X)pédiées. L'ensemble d'activités que nous possédons est donc décrit par l'ensemble $(\{D, P, S, M, E, T, X, A\})$, où A indique une « autre » activité. L'expression rationnelle exprimant cette succession d'activités est simple à construire :

$$(DP|PD)SM(E|TX).$$

Soit découpe puis peinture ou peinture puis découpe, suivi de soudure et montage, suivi de l'entreposage ou du tri avant expédition. Comme pour les règles sur les activités et les successions d'activités, pour déplacer la règle à la période désirée il suffit de rajouter au début de l'expression que l'on autorise à faire n'importe quelle activité k fois où k est la période où la règle s'applique. Ici nous voulons faire ces

activités la deuxième période. Posons $\sigma = D|P|S|M|E|T|X|A$. On écrit donc :

$$(\sigma)\{2\}(DP|PD)SM(E|TX).$$

La règle ne stipulant plus rien pour la suite de la journée, on autorise n'importe quelle activité. L'expression rationnelle représentant notre règle de construction est donc :

$$(\sigma)\{2\}(DP|PD)SM(E|TX)\sigma *.$$

La figure 5.13 est l'automate issu de cette expression.

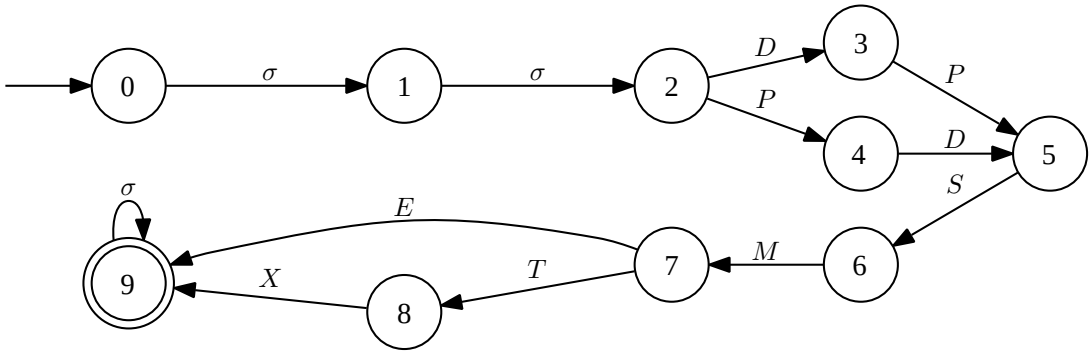


FIGURE 5.13 – Automate représentant une règle de construction de châssis automobile

Pour résumer, nous pouvons modéliser l'obligation de voir apparaître un motif complexe à une date fixée d'une manière simple. Il faut néanmoins que le motif à modéliser soit exprimable sous forme d'un automate fini ou d'une expression rationnelle. Soit m un tel motif, k la date à laquelle la règle s'applique et $\sigma = \bigcup_{a \in A} a$, l'union de l'ensemble des activités, alors l'expression rationnelle suivante décrit complètement la règle qui impose l'apparition du motif m à la $k^{\text{ème}}$ période de l'horaire :

$$\sigma\{k\}m\sigma *.$$

Cette expression, une fois transformée en automate, pourra être combinée à d'autres règles comme pour tous les automates que nous utilisons.

Interdire un motif. Interdire un motif à une date donnée, est, lorsque l'on sait comment le rendre obligatoire, trivial. Prenons par exemple le cas du travail en fin de semaine. Une règle qui est apparue souvent dans les problèmes de planification de personnel concerne les créneaux du vendredi, samedi et dimanche. Elle stipule qu'il ne peut y avoir ni repos, ni créneau de travail le matin avant un week-end chômé. Si l'on considère les activités travaillées $\{(M)\text{atin}, (S)\text{oir}, (N)\text{uit}\}$ ainsi que le (R)epos, nous souhaitons donc interdire, à partir du vendredi les motifs RRR et MRR. Un horaire d'une semaine suivant cette règle aura donc 4 journées non contraintes suivies de motifs n'étant ni RRR ni MRR. En reprenant le symbole σ représentant la disjonction de toutes les activités, on peut écrire l'expression rationnelle suivante :

$$\sigma\{4\} \sim (RRR|MRR).$$

Cette expression est-elle correcte pour décrire complètement un horaire d'une semaine respectant l'interdiction de motif sur la fin de semaine ? Dans ce cas particulier, elle le sera. En effet, nous pouvons

constater que les 4 premiers jours ne sont pas contraints, puis nous construisons le complémentaire de l'expression $(R|M)RR$. Nous nous heurtons ici à un problème concernant la taille du complémentaire. En effet, les motifs RR et $RRRR$ appartiennent au complémentaire de notre expression et, concaténés à $\sigma\{4\}$, ils pourraient former un motif interdit. Ceci n'arrivera pas dans ce cas précis, car la règle se situe en fin d'horaire. Les mots acceptés représentant un horaire valide ont une taille de 7, les 4 premiers créneaux étant définis, l'expression $\sim (RRR|MRR)$ aura forcément une taille 3. Si cela ne pose pas de difficulté à la résolution du problème que l'on modélise, on ne peut trouver satisfaisant de ne pas modéliser complètement la règle à travers cette expression. On pourrait par exemple imaginer que le modèle automate que nous construisons soit utilisé dans un cadre ne prenant pas en compte la taille de l'horaire. Le modèle serait alors erroné. De plus, si la règle était valide les mardi, mercredi et jeudi, la contrainte sur la taille de la règle pourrait être violée du fait de l'incertitude potentielle de la suite de l'expression. Par exemple, $\sigma \sim (RRR|MRR)\sigma*$ n'imposerait pas de taille au complémentaire de la règle qui serait alors mal modélisée. Nous devons pour se faire introduire un nouvel opérateur pour définir le complémentaire d'un motif pour une taille donnée.

Définition 29 (Complémentaire de taille k) *Soit m une expression rationnelle dans un alphabet Σ, on appellera complémentaire de taille k, l'ensemble des mots de taille k appartenant au complémentaire de m dans $\Sigma*$. Le complémentaire de taille k du motif m sera noté : $\sim_k m$.*

Le calcul du complémentaire de taille k d'un motif m se fait à l'aide de l'opération d'intersection et de complément :

$$\sim_k m = \sim m \cap \sigma\{k\}.$$

Cette opération étant maintenant définie, on peut maintenant exprimer de manière correcte l'interdiction d'apparition d'un motif d'une taille donnée dans un horaire. Dans notre exemple, l'expression rationnelle décrivant la règle,« ni matin ni repos avant un week-end chômé » devient par conséquent :

$$\sigma\{4\} \sim_3 (RRR|MRR).$$

La figure 5.14 représente l'automate construit à partir de cette expression.

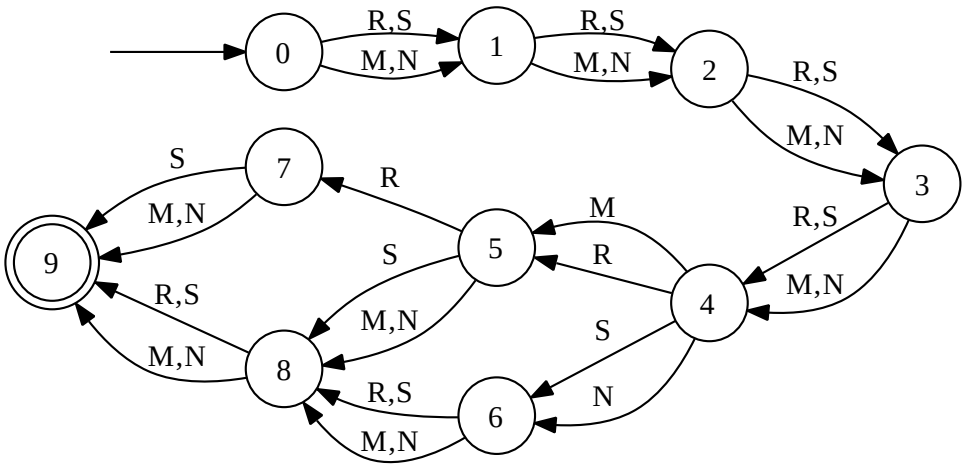


FIGURE 5.14 – Automate représentant la règle « ni matin ni repos avant un week-end chômé »

Si cet automate est correct, la méthode de construction automatique nous limite à l'interdiction de motifs d'une taille donnée à l'avance. Or, nous voudrions pouvoir imposer l'interdiction de motifs plus

complexes. À date fixe, interdire un motif rationnelle revient en fait à dire qu'à partir de cette date, aucun mot préfixé par ledit motif ne peut être formé. Or, représenter l'ensemble des mots préfixés par un motif m, connaissant l'ensemble des symboles autorisés σ, se fait par simple concaténation. Ainsi l'ensemble des mots commençant par le motif m s'écrit :

$$m\sigma * .$$

Le complémentaire de cette expression représente donc l'ensemble des mots non préfixés par le motif m. Finalement, interdire l'apparition du motif m à la date k s'écrit de manière générale à l'aide de l'expression rationnelle suivante :

$$\sigma\{k\} \sim (m\sigma *).$$

En reprenant l'exemple de la règle « ni matin ni repos avant un week-end chômé », le motif interdit est $RRR|MRR$, l'expression rationnelle permettant de construire l'automate d'après la formule générale est donc :

$$\sigma\{4\} \sim ((MRR|RRR)\sigma *).$$

L'automate figure 5.15 est la transformation de cette expression en automate. Pourquoi est-il différent de l'automate figure 5.14 ? Simplement parce que pour palier aux lacunes de notre première proposition de modélisation automatique, nous proposions de limiter la taille de la sous-expression concernée par la règle afin d'empêcher l'apparition de motif interdit. Cela nous obligeait à expliciter les 7 jours de l'horaire considéré. L'intersection de l'automate figure 5.15 avec l'automate acceptant tous les mots de taille 7 produit l'automate figure 5.16 en tous points similaires au précédent figure 5.14. Cette dernière modélisation offre donc un double avantage :

- une généralisation à tout type de motif rationnel ;
- des automates plus petits donc plus efficaces lors des opérations de regroupement de règles (intersection, complément, union).

Motif obligatoire sans date. Dans ce paragraphe, nous nous intéressons à la modélisation de règles imposant l'apparition d'un motif au cours de l'horaire, sans préciser la date d'apparition. Ce type de règle apparaît notamment lorsqu'une tâche, composée d'une succession d'activités, doit être effectuée dans une journée par exemple. Au cours d'une semaine de cours à l'université, on souhaite qu'un et un seul cours magistral d'informatique soit suivi dans cette même semaine d'un créneau de TD (travaux dirigés) ou d'un créneau de TP (travaux pratiques). Si l'on note M un cours magistral, D un TD, P un TP et $\sigma' = \sigma/\{M, D, P\}$ la disjonction de l'ensemble des activités disponibles aux étudiants n'appartenant pas aux trois première catégories, alors le motif modélisant la succession d'un cours magistral et d'un TP ou d'un TD est :

$$M\sigma' * (D|P).$$

Le $\sigma' *$ indique que la succession cours magistral, travaux dirigés ou pratiques n'est pas forcément immédiate. Maintenant, pour placer cette succession d'activités scolaires dans le planning d'une semaine, il suffit d'encadrer l'expression précédente par des σ' pour indiquer que l'on peut faire d'autres activités avant et après :

$$\sigma' * M\sigma' * (D|P)\sigma' * .$$

Nous produisons donc, pour cette règle d'apparition du motif « cours magistral suivi d'un TP ou d'un TD dans la semaine », l'automate figure 5.17.

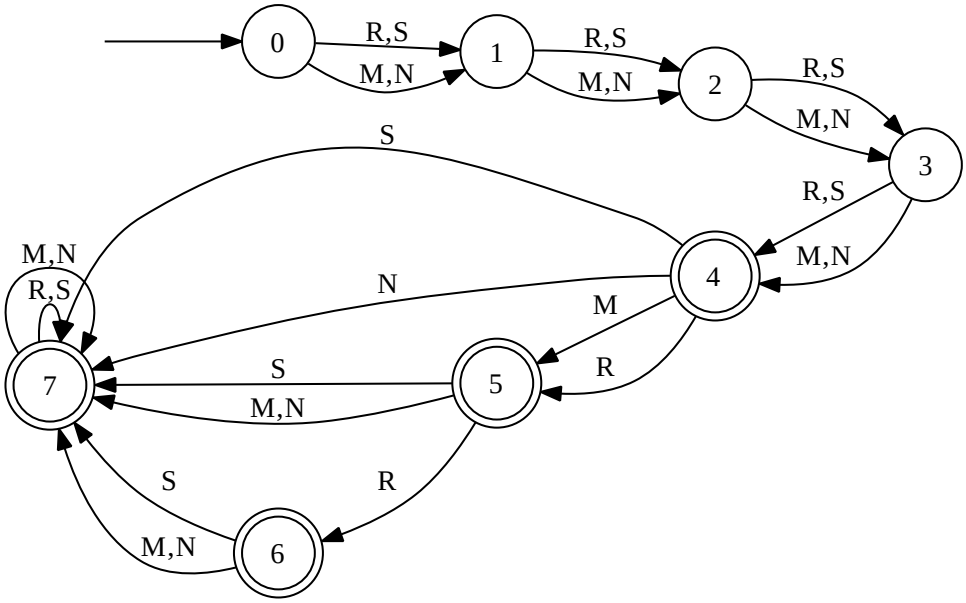


FIGURE 5.15 – Automate représentant la règle « ni matin ni repos avant un week-end chômé »

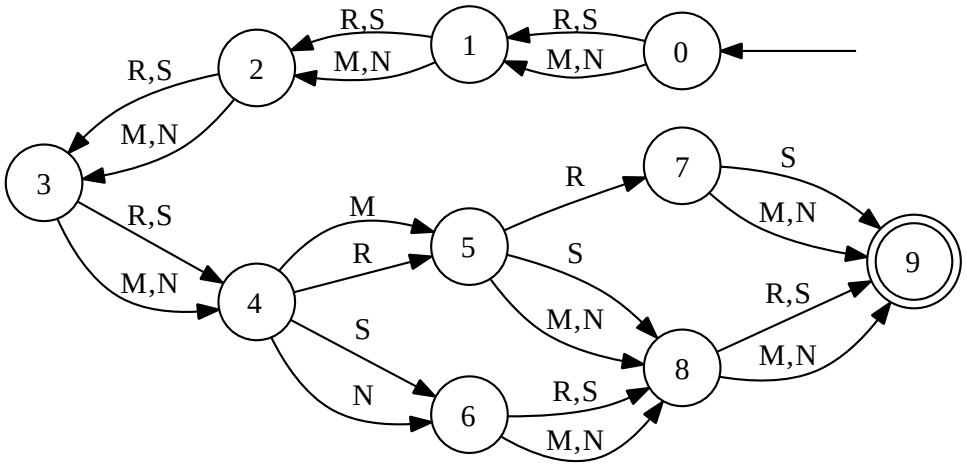


FIGURE 5.16 – Automate représentant la règle « ni matin ni repos avant un week-end chômé » dans un horaire de taille 7

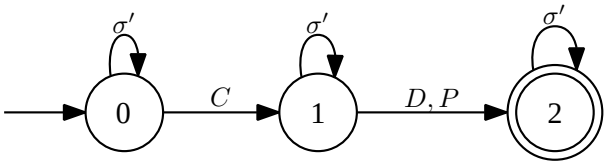


FIGURE 5.17 – Automate représentant la règle « un CM est suivi dans la semaine d'un TD ou d'un TP »

Cet exemple illustre bien la méthode que nous proposons pour construire un automate représentant une règle de séquencement dans laquelle un motif rationnel est imposé. Pour imposer l'apparition d'un motif m, nous construisons simplement l'expression rationnelle $\sigma * m\sigma *$. Ainsi l'horaire formé contiendra forcément le motif m. Notons que cette formulation ne présume pas du nombre d'apparitions du motif m. Nous imposons uniquement qu'il apparaisse au moins une fois. Une force des expressions rationnelles est leur capacité d'assemblage par les opérations de concaténation d'intersection et d'union. Ainsi, nous avons utilisé cette capacité dans notre exemple pour réduire simplement le problème de voir apparaître des activités dans un ordre donné mais sans contrainte sur l'immédiateté de la succession. En effet, dans notre exemple, le cours magistral et les travaux dirigés ou pratiques ont un ordre d'apparition précis mais il n'est pas demandé (et d'ailleurs déconseillé) d'enchaîner ces activités. Ainsi, le motif $M(D|P)$ ne représenterait pas correctement la règle. Ajouter entre le cours magistral et les TD/TP la possibilité de faire d'autres activités s'est fait naturellement en ajoutant σ' au milieu de l'expression rationnelle. Or, de par cette flexibilité et de la fermeture des langages rationnelles par la concaténation, $M\sigma'(D|P)$ est aussi un motif rationnel. L'apparition de ce motif dans un horaire répond donc également au cadre des motifs obligatoires.

L'automate figure 5.18 représente l'automate général pour les règles où un motif m doit apparaître au cours de l'horaire que l'on cherche à construire.

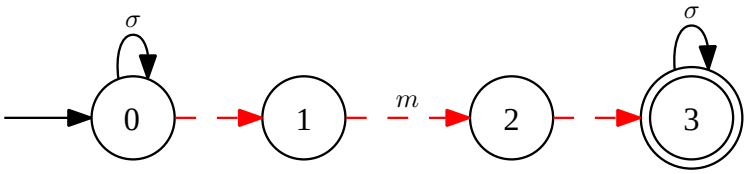


FIGURE 5.18 – Automate représentant l'obligation de voir le motif m apparaître au cours d'un horaire

Succession de motifs. Nous savons désormais comment imposer un motif donné, et même, en jouant sur la flexibilité des expressions rationnelles, faire en sorte qu'un motif suivi d'un second motif soit forcé d'apparaître. Néanmoins, modéliser ce type de règle grâce à l'expressivité des langages rationnels ne suffit pas toujours. Prenons par exemple, dans le cadre de la planification d'infirmières, la contrainte « un repos est obligatoire après deux nuits ». Cette dernière sera exprimée simplement par l'infirmière en chef et avec un peu de formation, on pourra lui demander de l'exprimer sous forme d'une suite d'activités. Il faut encore une fois définir un symbole pour chaque activité. Ici les activités sont jour J, nuit N et repos R. Exprimer la règle ci-dessus revient donc à poser l'expression rationnelle suivante :

$$NNR.$$

Cette expression rationnelle n'exprime bien entendu que la possibilité de faire un repos après deux nuits. Elle doit être complétée pour représenter l'ensemble des horaires valides pour cette règle. En effet, on peut traduire la règle par « si deux nuits sont faites alors elles sont suivies par un repos » ce qui implique que d'autres motifs peuvent apparaître durant l'horaire. Pour ce faire, il faut expliciter les cas autorisés, c'est-à-dire ne violant pas la règle. Les séquences autorisées sont une succession de jours ou de nuits $J|R$ ou une nuit suivie d'un jour ou d'un repos $N(J|R)$ ou alors notre règle deux nuits puis un repos NNR. Ainsi, l'expression rationnelle définissant la règle peut être construite comme une combinaison de ces séquences :

$$((J|R)|(N(J|R))|(NNR)) * .$$

Si l'expression rationnelle peut apparaître un peu obscure, l'automate équivalent figure 5.19 dérivé à partir de cette expression est plus explicite.

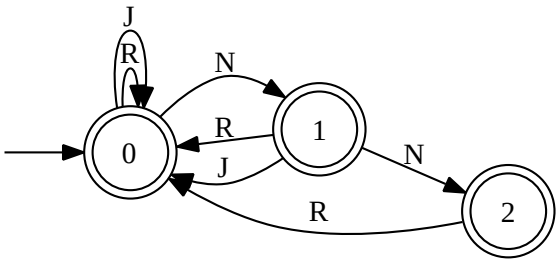


FIGURE 5.19 – Automate représentant la règle « un repos est obligatoire après deux nuits »

Il est possible de construire une telle expression rationnelle « à la main ». Néanmoins sous cette forme, il est délicat de la générer automatiquement. En effet, il faut pouvoir expliciter quelles sont les séquences autorisées, c'est-à-dire qui ne correspondent pas à la règle. De ce fait, il devient plus facile de travailler avec la forme négative de la règle. Si l'on reprend notre exemple la règle devient « il est interdit d'avoir autre chose qu'un repos après deux nuits ». Il est donc suffisant de connaître l'ensemble des activités qui ne sont pas un repos. Ici, il s'agit d'un jour ou d'une nuit. Ainsi, un horaire valide respectant cette règle est une succession d'activités qui ne forme pas « nuit-nuit-nuit » ou « nuit-nuit-jour ». Du point de vue de l'expression rationnelle on souhaite donc interdire NNJ et NNN. Ce qui se traduit par $NN(J|N)$. Concernant l'horaire, il s'agit donc d'interdire toute séquence où cette expression apparaîtrait. Il est donc interdit d'avoir n'importe quelle activité suivie de la séquence $NN(J|N)$ suivie de n'importe quelle activité. L'expression rationnelle décrivant la règle « un repos est obligatoire après deux nuits » est donc la négation de cette séquence :

$$\sim ((J|N|R) * (NN(J|N))(J|N|R)*).$$

Il suffit donc de connaître l'ensemble des activités pour construire cette expression automatiquement. L'ensemble des séquences autorisées est déduit automatiquement par l'opération sur l'automate consistant à trouver le complémentaire du langage qu'il définit.

La figure 5.20 présente l'automate construit à partir de l'expression rationnelle négative. Il est en tout point identique à celui présentant la forme positive de l'expression rationnelle figure 5.19. Le placement des états diffère car la génération automatique de l'automate ne produit pas toujours le même placement des états.

De manière systématique, on comprend alors qu'il suffit, pour forcer un motif exprimé par une expression rationnelle, de bien comprendre sa structure. En effet, une erreur commune lorsque l'on exprime une règle à l'aide d'expressions rationnelles est de ne pas saisir la notion de précédence d'un motif par rapport à l'autre. Lorsque nous écrivons NNR est un motif obligatoire, nous indiquons que **si** le motif NN apparaît **alors** il est **immédiatement** suivi par un repos. Cela ne veut en aucun cas dire que le motif NNR doit apparaître dans l'horaire de l'employé.

Ce constat nous amène à la généralisation suivante : Soit une expression rationnelle m qui, si elle est présente, doit être immédiatement suivie par un autre motif m'. Comment construire l'automate représentant cette règle (i.e. si m est rencontré alors m', et sinon n'importe quelle séquence) ? Dans cette généralisation, il faut bien faire attention. Construire l'expression $m(\sim m')$ ne représente pas un motif

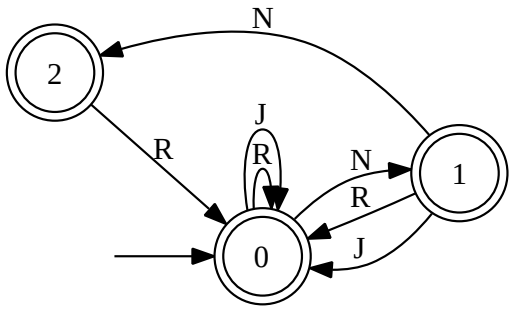


FIGURE 5.20 – Automate représentant la règle « un repos est obligatoire après deux nuits » par construction automatique

interdit. En effet, si l'on pose $m' = R$ alors $RRJ \in (\sim m')$. Pourtant la concaténation de m et $\sim m'$ ne devrait pas former de motif valide. Ainsi, un motif valide est en fait la concaténation de m et de m' suivi d'autres activités. Ce que nous voulons interdire est donc l'apparition de m suivi d'un motif qui ne commence pas par m'. Si l'on note σ n'importe quel symbole appartenant à l'alphabet (i.e. l'ensemble des activités) alors l'ensemble des mots qui ne commencent pas par le motif m' s'exprime à l'aide de l'expression rationnelle $\sim (m'\sigma*)$. Il ne reste plus qu'à concaténer cette expression avec la représentation des mots se terminant par le motif m, soit $\sigma * m$. Ainsi, par l'opération du complémentaire, nous pouvons construire l'expression rationnelle forçant le motif m' à suivre immédiatement le motif m s'il apparaît :

$$\sim (\sigma * m \sim (m'\sigma*)).$$

Cette construction d'expression rationnelle peut s'automatiser sous la forme d'un algorithme qui, à partir d'une règle de ce type, construit dans un premier temps les expressions rationnelles associées, puis l'automate représentant la règle. L'algorithme 4 réalise cette opération.

Entrées: A L'ensemble des activités, $m \to m'$ le motif obligatoire
Sorties: $\Pi_{\text{règle}}$, un automate forçant le motif si le préfixe apparaît
$\sigma \leftarrow \bigcup_{a \in A} a$;
$\Pi_\sigma \leftarrow \mathcal{A}(\sigma*)$;
$\Pi_m \leftarrow \mathcal{A}(m)$;
$\Pi_{m'} \leftarrow \mathcal{A}(m')$;
$\Pi_{\sigma m} \leftarrow \Pi_\sigma \cdot \Pi_m$;
$\Pi_{m'\sigma} \leftarrow \Pi_{m'} \cdot \Pi_\sigma$;
$\Pi_{\text{règle}} \leftarrow \sim (\Pi_{\sigma m} \cdot \sim \Pi_{m'\sigma})$;

Algorithm 4: Création automatique d'un automate de règle du type $m \to m'$

Compter un motif rationnel. Ce paragraphe présente un des plus intéressants challenges de cette thèse. En effet, au cours de notre étude sur les problèmes de planification de personnel, nous avons souvent rencontré des règles imposant qu'une séquence d'activités apparaisse un nombre de fois fixe ou non au cours de l'horaire à planifier. Dans le cadre hospitalier, certaines successions d'horaires sont tolérées un certain nombre de fois uniquement. Par exemple, le fait pour une infirmière de travailler le soir puis le matin ne peut être fait que deux fois par mois. Comment modéliser cette règle ? Sur une petite règle

possédant une limite faible (2), nous pourrions très bien modéliser cela par un automate. Le principe serait le suivant : à chaque fois que le motif SM est rencontré, changer d'état. Au bout de deux fois, interdire le motif SM. Cet exemple est illustré figure 5.21.

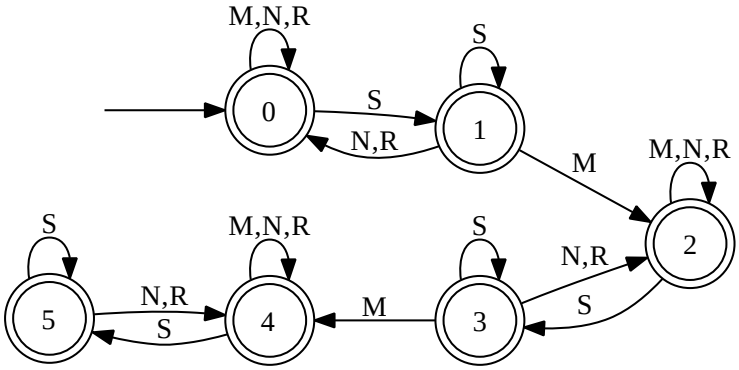


FIGURE 5.21 – Automate représentant la règle « pas plus de deux fois le motif soir-matin »

Que peut-on apprendre d'un tel exemple ? L'intuition, apportée par cet exemple, est que compter un motif rationnel à l'aide d'un automate fini déterministe simple, peut se révéler très coûteux en nombre d'états. Ici, nous devons gérer une règle de taille 2 et le nombre d'occurrences maximum de ce motif est de 2 également. Bien que ces tailles soient petites, nous obtenons un automate comportant 6 états et 23 transitions. Cette automate est encore de taille raisonnable. Néanmoins, si notre problème comporte un grand nombre de règles de ce type, l'intersection de toutes ces règles produira un automate de plus en plus gros. De plus, la construction de ce type d'automate n'est ni flexible, ni automatisable facilement. Il faut en effet être capable de déterminer à quelle partie du motif nous sommes pour décrire chaque transition, car même si l'on sort du motif que l'on souhaite compter, il n'est pas impossible que les activités forment à nouveau le début de ce motif. Il faut donc être capable de rediriger les transitions vers les états correspondant à ce motif partiel. Cela se réalise sans trop de difficulté, manuellement pour des motifs simples. Dès lors que le motif devient compliqué, il est quasiment impossible de construire un tel automate à la main sans erreur. De plus, l'automatisation devient complexe, il faut expliciter tous les cas possibles avant de minimiser l'automate le cas échéant.

Devant une automatisation complexe donnant des résultats peu satisfaisants, les automates pondérés semblent être ici une solution à la modélisation de règles comptant des motifs d'activités. En effet, ils permettent d'éviter que la complexité de l'automate soit dépendante de la borne du compteur. Ainsi, un automate comptant un motif d'activités aura la même taille quel que soit le nombre d'occurrences que l'on souhaite obtenir. Malheureusement, la construction automatique de ces automates n'est de prime abord pas plus simple que pour un automate classique. Il faut en effet trouver un moyen d'ajouter un coût de 1 à chaque fois que le motif est effectué. Intuitivement, il suffirait de porter un coût de 1 à la dernière transition du motif que l'on souhaite compter. Or, cela n'est pas trivial puisqu'une transition n'est identifiée que par le symbole qu'elle porte et ne présume pas des transitions l'ayant précédée. Comment alors être sûr qu'une transition donnée correspond bien à la dernière étape d'un motif ? Supposons que nous voulions compter le motif rationnel m. Il nous faudrait d'abord être capable de le reconnaître dans un automate. Il existe pour reconnaître des motifs dans du texte des algorithmes très efficaces tel l'algorithme d'Aho-Corasick [2]. Cependant nous avons tenté sans succès d'adapter cette technique à un automate qui peut être vu comme un ensemble de textes. Nous avons alors tenté une approche construc-

tive. Étant donné ce motif, pouvons-nous construire un automate pondéré directement ? Encore une fois la réponse nous vient des expressions rationnelles et consiste en une astuce simple et efficace. Si nous ne pouvons pas caractériser ce qui précède une transition, nous pouvons par contre connaître la transition suivante très facilement. L'idée est donc de modifier le motif m de manière à pouvoir identifier de manière certaine la ou les dernières transitions du motif. Soit Σ l'alphabet, comprenant tous les symboles possibles permettant de former un mot. L'équivalent en terme de planification serait l'ensemble des activités disponibles pour former un horaire. Soit α un symbole n'appartenant pas à l'alphabet : $\alpha \notin \Sigma$ et $m' = \alpha*$ l'expression rationnelle signifiant 0 ou plus α. Si α appartenait à l'alphabet alors il suffirait d'appliquer la formule que nous avons développée pour les successions de motifs : si m apparaît, alors il doit être suivi par le motif m'. Or, α n'appartenant pas à Σ, l'opération de complémentaire appliquée deux fois dans l'algorithme 4 ferait disparaître le symbole α. Il nous faut donc penser à une autre approche permettant de garder la symbolique d'α.

Nous cherchons à construire un automate, acceptant tous les mots formés par les symboles d'un alphabet Σ ainsi que le motif m' de manière explicite. Les mots construits seraient donc formés d'une succession de n'importe quel symbole de l'alphabet, suivi du motif m' zéro fois ou plus. En notant σ la disjonction des symboles de Σ, nous pouvons en déduire l'expression rationnelle suivante :

$$(\sigma * (m\alpha*)*)* \tag{5.1}$$

Cette expression résume parfaitement ce que nous souhaitions exprimer puisque si le motif m est rencontré, une fois ou plus, il sera suivi du motif $\alpha*$.

Reprenons l'exemple où l'on souhaite compter le nombre de fois que le motif SM apparaît. Nous posons $m = SM$ et $\sigma = M|N|R|S$. Regardons le résultat de l'automate produit par l'expression 5.1 sur la figure 5.22. Conformément à ce que nous attendions, si une transition marquée α est empruntée, alors le motif SM a forcément été lu par l'automate. Par conséquent, toutes les transitions ayant pour destination un état qui serait l'origine d'une transition marquée α correspondent à la fin du motif que l'on souhaite reconnaître. Ainsi, attribuer un coût de 1 à ces transitions permet de construire un automate pondéré comptant le nombre d'occurrences d'un motif.

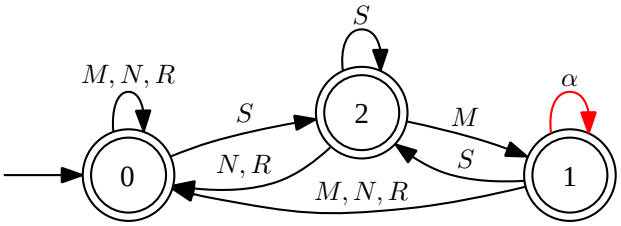


FIGURE 5.22 – Automate permettant de reconnaître le motif SM

La construction de notre automate pondéré n'est pas pour autant terminée. Il faut en effet retirer les transitions α qui ne correspondent à aucune activité. Puisque nous avons concaténé le motif $\alpha*$ et non α seul, la transition peut être retirée sans incidence sur la signification de l'automate. Ceci est dû à la signification de la fermeture de Kleene, ici : α apparaît zéro fois ou plus. Supprimer les transitions marquées α revient donc à imposer que le symbole α apparaisse zéro fois. Il suffit donc d'ajouter un compteur associé aux coûts ajoutés aux transitions menant à α pour obtenir un automate pondéré représentant la règle « pas plus de deux fois le motif soir-matin ». Comme précédemment, nous ajoutons une variable $y = [\![0, 2]\!]$

associée à l'automate pondéré que nous générons. Le résultat de ces transformations est montré figure 5.23.

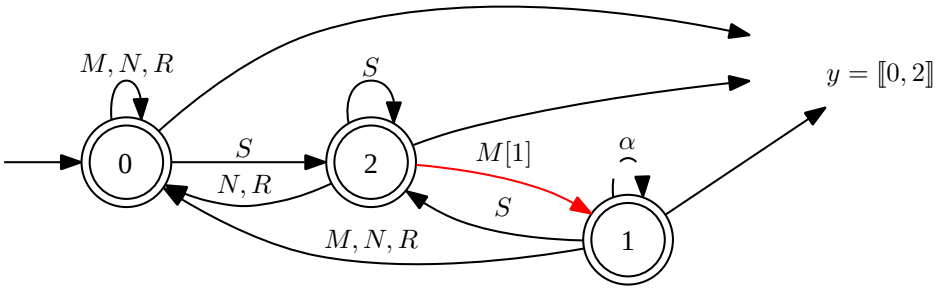


FIGURE 5.23 – Automate à compteur pour le motif SM

L'algorithme 5 permet d'automatiser la création d'un automate pondéré afin d'imposer des règles de comptage sur des motifs rationnels.

Entrées: A l'ensemble des activités, m le motif obligatoire à compter, l et u les bornes du nombre d'occurrence de m
Sorties: Π_m, un automate pondéré pour le motif m
$\sigma \leftarrow \bigcup_{a \in A}$ // Construire l'union des symboles;
$\Pi_m \leftarrow \mathcal{A}((\sigma * (m\sigma*)*)*)$ // Construire l'automate à partir de la formule 5.1;
pour tous les *états* $q \in \Pi_m$ **faire**
  si *la transition* $t = \delta(q, \alpha)$ *existe* **alors**
    Ajouter un coût de 1 pour la dimension r à toutes les transitions entrant q;
    Supprimer la transition t;
  fin
fin
Créer la variable $y_r = [\![l, u]\!]$ associée aux coûts de dimension r dans l'automate;

Algorithm 5: Création automatique d'un automate pour compter les occurrences du motif m

5.3 Règles de séquencement souples

Les méthodes de modélisation automatique que nous proposons permettent de représenter un grand nombre de règles de séquencement. Cependant, les nombreuses règles métiers rencontrées ne sont pas aussi strictes que celles que nous avons décrites.

Définition 30 (Règle de séquencement souple) *Une règle de séquencement est dite souple si elle peut être ignorée ou pénalisée. Une règle de séquencement souple est associée à un coût caractérisant l'importance de la violation.*

De la même manière que pour les règles de séquencement simples, nous caractérisons les différentes règles de séquencement souples et nous proposons une méthode de modélisation automatique de ces règles.

5.3.1 Activités

Les règles souples concernant une activité au cours d'une séquence sont similaires à celles rencontrées précédemment. On distinguera trois types de règles :

- interdire une activité ;
- forcer une activité ;
- compter une activité.

Chacune de ces règles pourra être effective à une date donnée ou sur toute une période.

Interdire une activité pendant la séquence. Dans le cas où l'interdiction d'une activité est obligatoire, nous avons vu qu'il était suffisant d'exprimer une inégalité entre la variable représentant l'activité effectuée au temps t et la valeur correspondant à l'activité. Ici, nous souhaitons interdire une activité de manière souple. C'est-à-dire que son apparition n'entraîne pas l'invalidité de la séquence construite mais une pénalité à quantifier.

Afin d'évaluer la pénalité, il nous faut d'abord connaître le nombre de fois où l'activité interdite apparait. L'utilisation d'un automate pondéré permet de compter ce nombre d'apparitions. Les automates et algorithmes présentés section 5.2.1 qui permettent de limiter ou d'imposer le nombre d'occurrences d'une activité peuvent être utilisés pour compter le nombre d'apparitions en supprimant les bornes de la variable compteur associée. Par la suite, il suffit de lier la variable compteur à une variable de coût pour pénaliser l'apparition de l'activité interdite.

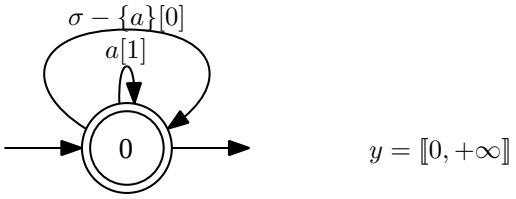


FIGURE 5.24 – Automate comptant l'activité a

Notons que dans le cas particulier de l'interdiction d'une activité, l'utilisation d'un automate est quelque peu abusive. Si cette représentation indique clairement que l'on va compter le nombre d'apparitions d'une activité a, l'absence de limite sur cette apparition la rend quelque peu lourde. En effet, il suffirait d'associer un coût à l'affectation de toute variable à la valeur a pour obtenir le même résultat.

Cependant, la modélisation sous forme d'automate autorise toujours, au-delà du caractère « parlant » de cette modélisation, de regrouper cette règle avec d'autres plus complexes. Cet avantage sera par ailleurs étudié dans le prochain chapitre.

Interdire une activité sur une période donnée $[\![t_1, t_2]\!]$. Lorsque nous souhaitions interdire l'apparition d'une activité a sur une période, la méthode consistait à retirer la possibilité d'affecter a aux variables correspondantes. Ici, nous souhaitons que cette règle soit souple, il ne faut donc pas retirer la possibilité d'effectuer l'activité a.

La solution consiste donc à compter le nombre d'apparitions de l'activité a sur la période souhaitée.

Nous proposons deux méthodes :

- On associe un coût à l'affectation de chaque variable correspondant à la période où la règle s'applique.
- On construit un automate pondéré tel celui figure 5.24 en associant la matrice de coût d'affectation suivante :

$$\forall t \in [\![0, T]\!], c_{tw} = \begin{cases} 1 \text{ si } w = a \text{ et } t \in [\![t_1, t_2]\!] \\ 0 \text{ sinon} \end{cases}$$

Le coût de 1 permet de compter le nombre d'occurrences de l'activité a dans l'horaire. Cette méthode permet des pénalisations plus complexes comme par exemple un coût de 1 si a apparaît une seule fois, 10 sinon. Notons que ce type de pénalisation est très présent dans les problèmes de planification de personnel où certaines règles expriment des pénalisations plus importantes plus l'activité apparait.

5.3.2 Succession d'activités

Que ce soit pour une activité simple ou une succession d'activités, la problématique des règles souples reste la même. Il faut pouvoir compter le nombre de fois où la règle n'est pas respectée. Dans le cadre des successions d'activités, nous avons proposé l'automate permettant de compter le nombre de successions de longueur $n = [\![l, u]\!]$ d'un même ensemble d'activités. L'algorithme 3 produisant cet automate figure 5.12 reste valable dans le cas de contraintes souples. Afin de pénaliser l'apparition d'une succession d'activités de longueur $n = [\![l, u]\!]$, il est suffisant de lier la variable compteur y de l'automate figure 5.12 à une variable de coût z pénalisant les occurrences de la succession.

Ce problème, ramené à une date fixe, peut être interprété de deux manières différentes :

- pénaliser une succession d'activités commençant à la date t ;
- pénaliser une succession d'activités qui se termine à une date ou période donnée.

Dans le premier cas, il faut déplacer l'état initial de l'automate en permettant toute activité jusqu'au temps t. Les transitions sortantes quant à elles, pénalisées par un coût de 1, rejoignent un état terminal comme sur l'automate figure 5.9.

Dans le second cas, il suffit de modifier la matrice de coût d'affectation de manière à ne pas pénaliser les arcs sortants de la succession d'activités pendant la période. Comme la matrice d'affectation est dépendante de la date, ce changement est trivial.

Utiliser ces méthodes permet d'éluder le cas où l'on pénalise les successions d'activités en fonction de leur apparition et non de leur taille. Or, il semble intéressant de pouvoir pénaliser une succession d'activités en fonction de sa longueur.

Il ne s'agit plus de compter l'apparition d'une séquence. Nous n'utiliserons donc plus de variable compteur, mais directement une variable de coût permettant de connaître directement le coût du chemin construit.

Supposons que nous voulions pénaliser les successions de l'activité a à partir d'une longueur l jusqu'à une longueur u de manière à ce que chaque fois que la longueur de la succession augmente on double le coût de la séquence. Ainsi, une succession de longueur l coûtera 1, $l + 2$ 4, $l + 4$ 16.... Pour généraliser, nous utiliserons une fonction $f(n)$ qui, à la longueur de la succession d'activités n, associe son coût. Dans le cas où la pénalisation double à chaque répétition, nous posons :

$$\forall n \in [\![0, T]\!], f_n = \begin{cases} 2^{n-l} \text{ si } n \in [\![l, u]\!] \\ 0 \text{ sinon} \end{cases}$$

L'automate se construit à l'aide d'une simple modification de l'algorithme 3, où l'on remplace l'affectation du coût de 1 par $f(i)$ (Algorithme 6).

Entrées: Une activité a, des bornes l et u de la longueur de la succession, une fonction $f : \mathbb{N} \to \mathbb{N}$
Sorties: Un automate pénalisant les successions d'activités de longueur $n \in [\![l, u]\!]$, une variable de coût z représentant cette pénalité.
pour tous les $i \in [\![0, u]\!]$ **faire**
  Créer un nouvel état final q_i
fin
pour tous les $i \in [\![0, u-1]\!]$ **faire**
  Ajouter la transition $q_i \xrightarrow{a} q_{i+1}$ de coût 0;
fin
Créer une nouvelle variable $z = [\![0, +\infty]\!]$;
pour tous les $i \in [\![l, u]\!]$ **faire**
  Ajouter la transition $q_i \xrightarrow{\sim a} q_0$ de coût $f(i)$;
fin
Ajouter la transition $q_0 \xrightarrow{\sim a} q_0$ de coût 0;

Algorithm 6: Création d'un automate pénalisant les successions d'activités au-delà d'une certaine taille

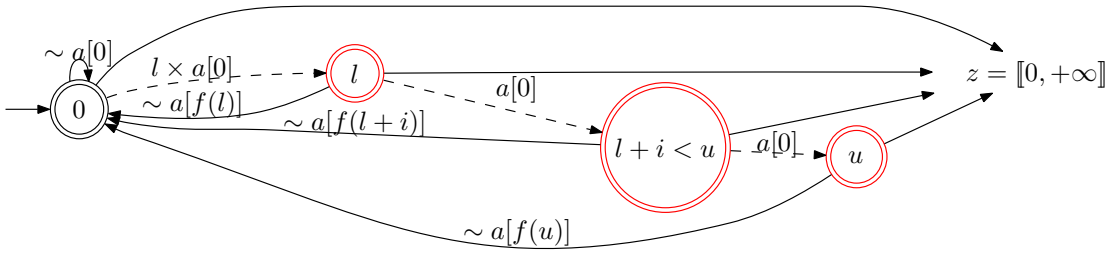


FIGURE 5.25 – Automate pénalisant les successions de l'activité a en fonction de la longueur

De la même manière que précédemment, rendre cette règle opérante uniquement à une date ou période fixée est possible, soit en décalant le début de la règle en forçant toute activité jusqu'à la date souhaitée, soit en modifiant la fonction f de manière à affecter un coût différent à la période recherchée.

5.3.3 Motifs d'activités

Dans la section 5.2, nous avons introduit différentes méthodes et algorithmes pour construire des automates modélisant des règles variées portant sur des activité simples, des successions d'activités et finalement des motifs rationnels d'activités. Ce dernier type nous a amené à développer une méthode automatisée pour compter le nombre d'apparitions d'un motif donné.

Or, rendre souple une règle concernant l'apparition d'un motif consiste à pénaliser l'apparition de ce motif. L'algorithme 5 permet de construire un automate comptant le nombre d'apparitions. Associer une variable de pénalisation à la variable compteur de l'automate pondéré permet donc de rendre souple les règles d'apparitions de motifs.

5.4 Conclusion

Dans ce chapitre, nous avons montré qu'un grand nombre de règles de séquencement peuvent être modélisées à l'aide d'automates. Nous avons montré le caractère expressif et compact des automates pour modéliser des règles de séquencement complexes dans le cadre de problèmes de planification. Les automates finis, pondérés ou non, offrent donc des outils très puissants de modélisation de ces règles.

Un autre avantage, que nous n'avons pas encore exploré, est la fermeture de ces automates devant des opérations usuelles, telles la concaténation, l'union ou l'intersection. Ce sera l'objet du chapitre suivant qui illustrera l'application de ces opérations lorsque nous possédons plusieurs règles de séquencement à traiter.

Chapitre 6

Agrégation de règles de séquencement

Sommaire

DANS le chapitre précédent, nous avons montré que les automates finis, pondérés ou non, sont des outils très puissants pour modéliser des règles parfois complexes de séquencement dans le cadre des problèmes de planification. Intéressons-nous maintenant à la fermeture de ces automates devant des opérations usuelles telles la concaténation, l'union ou l'intersection. Dans ce chapitre, nous nous attacherons tout d'abord à montrer l'intérêt de ces propriétés, puis nous rappellerons de manière exhaustive les opérations pour lesquelles un langage rationnel est fermé. Nous décrirons ces opérations de manière algorithmique. Enfin nous présenterons un algorithme d'intersection pour les automates pondérés permettant de rester dans le cadre des langages rationnels.

6.1 Intérêt

Un grand nombre de règles de séquencement peuvent être modélisées à l'aide d'automates. Lorsque nous modéliserons un problème de planification de personnel, nous créerons ces automates à l'aide des

algorithmes fournis au chapitre 5. Chaque automate représente pour chaque règle l'ensemble des séquences autorisées par la règle. Cette information est très importante. Néanmoins, lorsque nous possédons plusieurs règles, ne peut-on pas faire plus de déductions ? La réponse que nous avons choisie repose sur la propriété des langages rationnels à rester rationnels lorsque l'on applique sur eux des opérations de complément, d'intersection, d'union ou de concaténation. Nous détaillerons ces opérations dans la section suivante 6.2. Les grammaires supérieures aux grammaires rationnelles, les grammaires non contextuelles ne possèdent pas de telles propriétés. De notre point de vue, cela rend très délicat la modélisation de règles à l'aide de ce type de grammaires. Il faut en effet, d'une part être capable d'écrire des règles de production de la grammaire et ce de manière parfois récursive, et d'autre part il faut, si l'on souhaite agréger des règles, réécrire complètement la grammaire sans garantie que le résultat soit une grammaire non contextuelle [53].

6.1.1 Un premier exemple

Un des défis proposés dans les problèmes de planification de personnel est l'hétérogénéité des règles présentes. Non seulement de par leur forme, mais de par leur origine, les règles peuvent, par exemple, provenir d'une contrainte du code du travail, tout comme de la politique particulière à l'organisation concernée. Supposons qu'une première règle r_1 indique qu'une séquence de travaux, choisis parmi les activités (N)uit, (J)our, (R)epos, impose de commencer, soit par une séquence de trois nuits, soit par une séquence de trois repos.

L'expression rationnelle consacrée pour cette règle peut se construire en suivant les méthodes de génération d'automates présentées chapitre 5.

Nous obtenons en gardant la même sémantique que précédemment (i.e. $\sigma = J|N|R$) :

$$(RRR|NNN)\sigma *.$$

Nous pouvons construire l'automate fini déterministe équivalent figure 6.1.

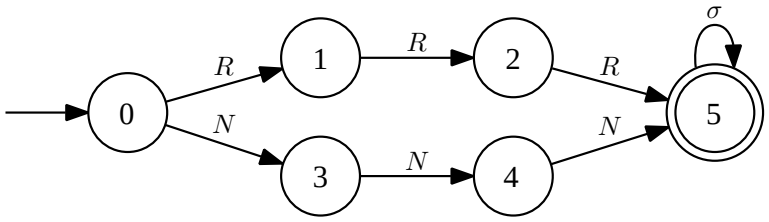


FIGURE 6.1 – Automate représentant la règle r_1 « trois repos ou trois nuits en début d'horaire »

De la même manière, construisons la règle r_2 imposant soit trois jours, soit trois nuits en début d'horaire. La construction de l'expression rationnelle $(JJJ|NNN)\sigma*$ et de l'automate associé figure 6.2 se fait de manière identique que précédemment.

Nous avons maintenant dans notre ensemble de règles, r_1 et r_2 que nous souhaitons imposer. De manière naturelle, sur cet exemple, on comprend qu'imposer ces deux règles en même temps revient à imposer le fait d'avoir trois nuits en début d'horaire, soit l'expression rationnelle suivante :

$$NNN\sigma *.$$

L'automate figure 6.3 est équivalent à cette expression rationnelle.

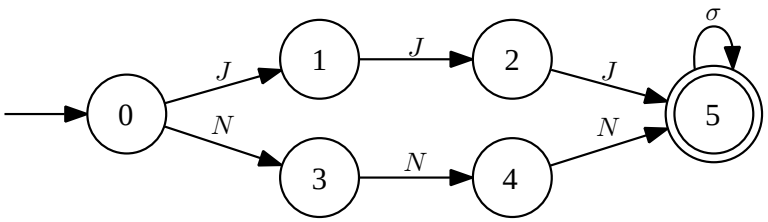


FIGURE 6.2 – Automate représentant la règle r_2 « trois jours ou trois nuits en début d'horaire »

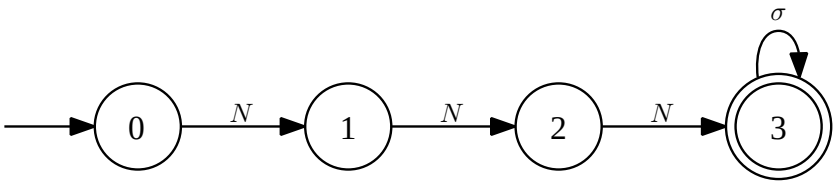


FIGURE 6.3 – Automate représentant la conjonction des règles r_1 et r_2 « trois nuits en début d'horaire »

Nous pouvons donc remplacer les deux automates figures 6.1 et 6.2 comportant chacun 6 états, par l'automate figure 6.3 comportant 4 états. Cet exemple illustre bien l'intérêt d'agréger les règles. Si cette simplification, dans le cadre de règles aussi simples, est triviale, elle peut devenir plus complexe lorsque nous traiterons des règles comportant des séquencements issus d'expressions rationnelles plus compliquées. Comment alors générer automatiquement l'agrégation de deux règles ? L'algèbre de la théorie des automates nous permet de répondre à cette question. Lorsque nous souhaitons imposer la règle r_1 et la règle r_2, nous imposons en fait la règle

$$r_{1 \text{ et } 2} = r_1 \wedge r_2.$$

Or, dans la théorie des automates, l'opérateur $\wedge$ est l'intersection d'automates. La présentation des algorithmes usuels des automates section 6.2 démontrera que l'intersection des automates représentant les règles r_1 et r_2 produit exactement l'automate $r_{1 \text{ et } 2}$.

6.1.2 Un exemple plus complexe

Soit un ensemble de règles portant sur l'horaire d'un employé issu de l'énoncé suivant : « au moins deux nuits pendant l'horaire ou au moins trois jours mais la première règle ne doit pas s'appliquer. » Nous pouvons identifier deux règles dans cet énoncé :

- r_1 : au moins deux nuits ;
- r_2 : au moins trois jours.

Modélisons ces règles sous forme d'expressions rationnelles puis d'automates (figures 6.4 et 6.5) :

$$\begin{aligned} r_1 &\equiv (\sigma - \{N\})^* N (\sigma - \{N\})^* N \sigma^*; \\ r_2 &\equiv (\sigma - \{J\})^* J (\sigma - \{J\})^* J (\sigma - \{J\})^* J \sigma^*. \end{aligned}$$

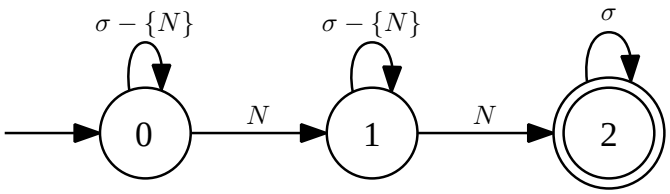


FIGURE 6.4 – Automate représentant la règle r_1 « au moins deux nuits pendant l'horaire »

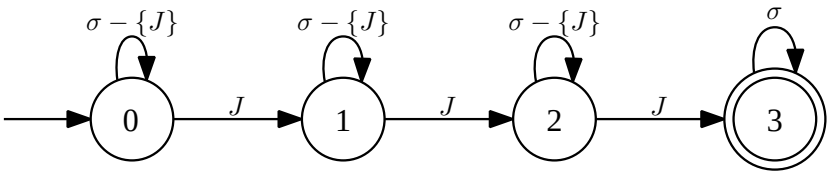


FIGURE 6.5 – Automate représentant la règle r_2 « au moins trois jours pendant l'horaire »

La difficulté ici consiste à interdire la règle qui n'est pas appliquée. En effet, si l'on utilise r_2, alors l'énoncé dit qu'on ne peut pas avoir r_1. Mais si l'on utilise r_1, malgré l'absence de précision, on ne peut pas utiliser r_2, car dans le cas contraire cela nous interdirait par définition r_1 ce qui entre en contradiction avec l'utilisation de r_1. L'énoncé correct s'exprime donc comme la disjonction exclusive des règles r_1 et r_2 :

$$r_1 \oplus r_2.$$

Nous ne savons pas construire cette expression à proprement parler. Cependant, la logique booléenne nous donne la formule suivante :

$$r_1 \oplus r_2 \equiv (r_1 \wedge \sim r_2) \vee (\sim r_1 \wedge r_2).$$

La construction des expressions rationnelles $\sim r_1$ et $\sim r_2$ n'est pas triviale, cependant la construction du complémentaire d'un automate l'est. Il suffit pour un automate complet[1] d'inverser les états finals et non-finals.

Concernant la règle $\sim r_1$ l'automate équivalent se construit donc en reprenant l'automate figure 6.4 et en inversant les états finals. On obtient l'automate figure 6.6 en retirant les transitions menant uniquement à des états non-finals.

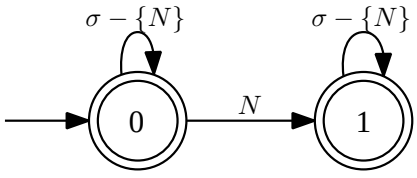


FIGURE 6.6 – Automate représentant l'opposé de la règle r_1

[1] on rappellera qu'un automate complet est un automate dont tous les états ont une transition sortante pour chaque symbole de l'alphabet

De façon similaire, nous pouvons construire l'automate représentant la règle $\sim r_2$ figure 6.7.

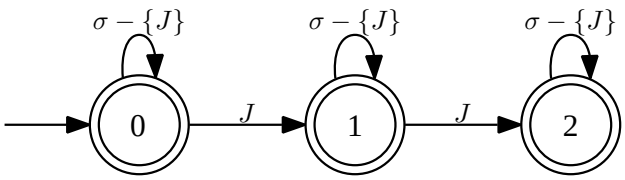


FIGURE 6.7 – Automate représentant l'opposé de la règle r_2

Comme pour l'exemple section 6.1.1, nous pouvons maintenant construire l'automate représentant la conjonction de la règle r_2 représenté par l'automate figure 6.5 et la règle $\sim r_1$ équivalente à l'automate figure 6.6. A l'aide de l'opération d'intersection on obtient l'automate figure 6.8. Cet automate est encore suffisamment simple pour remarquer qu'un état final n'est accessible qu'après au moins trois transitions marquées par un jour (J) et au plus une transition marquée par une nuit (N).

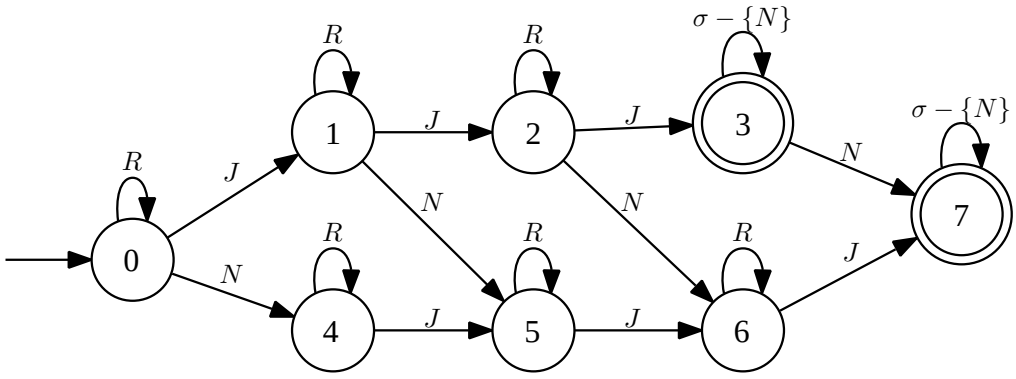


FIGURE 6.8 – Automate représentant l'intersection de la règle $\sim r_1$ et r_2

De la même manière, nous construisons l'automate représentant l'expression $\sim r_2 \wedge r1$. Le résultat est présenté figure 6.9.

Nous avons pu modéliser à l'aide d'automates les expressions $\sim r_1 \wedge r_2$ et $r_1 \wedge \sim r_2$. Pour terminer la transformation de l'énoncé « au moins deux nuits pendant l'horaire ou au moins trois jours mais la première règle ne doit pas s'appliquer », il faut exprimer la disjonction de ces deux expressions. Nous allons pour cela utiliser l'opération d'union des automates. L'automate figure 6.10 représente l'union des automates figures 6.8 et 6.9.

Pour construire un automate représentant une règle complexe, nous avons utilisé les opérations sur les automates suivantes :

- intersection ;
- complementaire ;
- union ;
- déterminisation ;

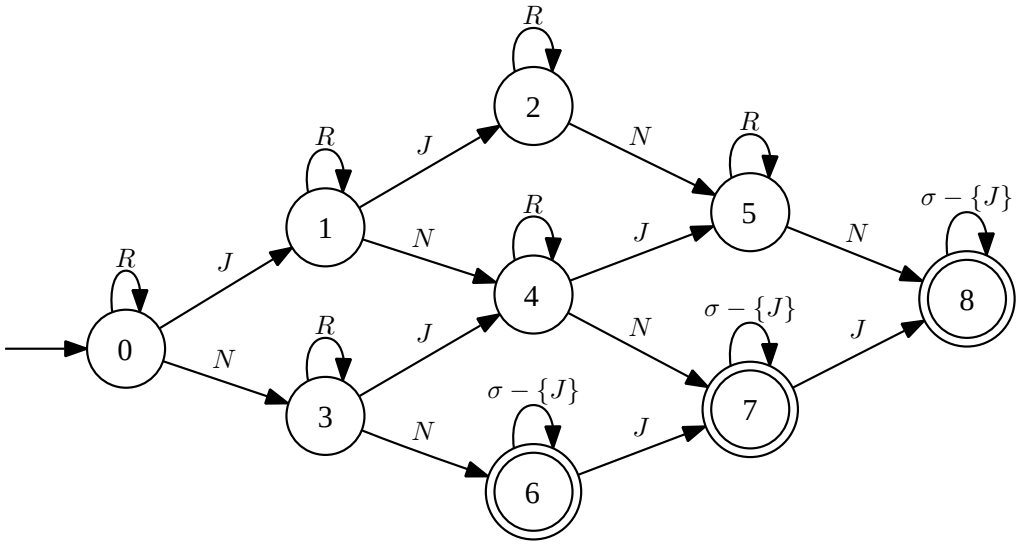


FIGURE 6.9 – Automate représentant l'intersection de la règle $\sim r_2$ et r_1

- minimisation.

Ces opérations sont présentes dans la majorité des bibliothèques permettant de manipuler des automates. L'intérêt d'utiliser ces opérations apparaît clairement au travers des deux exemples que nous avons proposés. Tout d'abord, cela permet parfois d'obtenir un automate plus petit représentant un ensemble de règles. Mais cela permet également de représenter des règles plus complexes en les construisant à l'aide d'un modèle décomposé. Pourquoi cela fonctionne ? Contrairement aux grammaires de niveau supérieur, les langages rationnels ont la propriété d'être stables lorsque l'on applique sur eux un ensemble d'opérations. Nous décrivons dans la section suivante les différentes opérations usuelles existantes, ainsi que les algorithmes et leurs complexités.

6.2 Opérations usuelles sur les automates finis

Lorsque nous avons fait le choix de nous intéresser à la modélisation des problèmes de planification de personnel à l'aide des grammaires formelles, nous avons cherché à trouver l'équilibre entre la puissance de modélisation et la facilité de modélisation. Si une grammaire non-contextuelle peut modéliser des règles plus complexes, nous avons considéré, que demander à un opérationnel d'écrire des règles de production d'une telle grammaire n'était pas envisageable. Au contraire, dans le chapitre précédent, nous avons montré qu'à partir de primitives simples, nous pouvions à l'aide des grammaires rationnelles, modéliser automatiquement un grand nombre de règles présentes dans les problèmes de planification de personnel. Au-delà de l'aspect fonctionnel, ce choix s'est fait en fonction de la souplesse qu'apporte les grammaires rationnelles. Nous pouvons, en effet, très simplement ajouter des règles, ou simplement les interdire, car l'intersection, l'union, la concaténation ou le complément d'un langage rationnel est un langage rationnel. Au contraire, pour les grammaires d'ordre supérieur cette propriété n'est pas garantie [53].

Alors que dans nos applications nous allons utiliser toutes ces opérations, nous allons dans cette section les décrire et les illustrer.

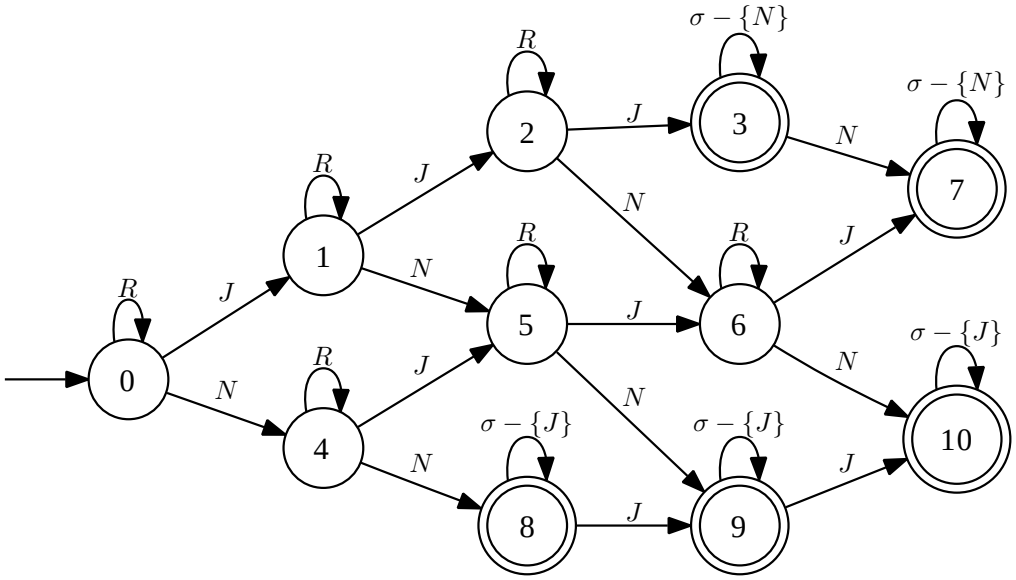


FIGURE 6.10 – Automate représentant l'union des automates figures 6.8 et 6.9 soit $r_1 \oplus r_2$

6.2.1 Complément

L'opération de complément pour un langage rationnel donné $\mathcal{L}$ consiste à construire le langage $\sim \mathcal{L}$, ensemble des mots n'appartenant pas à $\mathcal{L}$ dans le même alphabet. On écrit :

$$\sim \mathcal{L} = \Sigma^* - \mathcal{L}.$$

Autrement dit,

$$\sim \mathcal{L} = \{m \in \Sigma^* \mid m \notin \mathcal{L}\}.$$

Nous avons décrit pour l'exemple de la section 6.1.2 la construction d'un tel complément. Elle se fait en inversant les états finals et non-finals si l'automate est complet et déterministe. Il faut sinon rajouter les transitions manquantes.

Soit $\mathcal{A} = (Q, \Sigma, E, I, T)$ un automate fini, l'algorithme 7 construit l'automate complémentaire.

L'opération de complément s'effectue donc dans le pire des cas, si l'on suppose que l'on peut vérifier l'existence d'un membre de E en $O(1)$, en $O(|Q| \times |\Sigma|)$.

Montrons maintenant que le langage reconnu par l'automate complémentaire de $\mathcal{A}$ est un langage rationnel.

Théorème 1 (Complémentaire d'un langage rationnel) *Si $\mathcal{L}$ est un langage rationnel d'alphabet Σ, alors $\sim \mathcal{L} = \Sigma^* - \mathcal{L}$ est également un langage rationnel*

Preuve. Soit $\mathcal{L} = \mathcal{L}(\mathcal{A})$ pour un afd $\mathcal{A} = (Q, \Sigma, E, I, T)$. Soit $\sim \mathcal{L} = \mathcal{L}(\mathcal{B})$ où $\mathcal{B} = (Q, \Sigma, E, I, Q - T)$ c'est-à-dire le même automate que $\mathcal{A}$ en inversant les états finals. Ainsi un mot m est dans $\mathcal{L}(\mathcal{B})$ si et seulement si il existe un ensemble de transitions E_m inE menant de l'état initial à un état de $Q - T$. Si tel est le cas, alors m n'est pas dans $\mathcal{L}(\mathcal{A})$. □

Notons que calculer le complémentaire d'un automate fini non déterministe n'est pas trivial. Il faut pour cela le transformer en un automate fini déterministe.

Entrées: Un automate $\mathcal{A} = (Q, \Sigma, E, I, T)$
Sorties: Un automate $\mathcal{A}' =\sim \mathcal{A}$ acceptant uniquement l'ensemble des mots refusés par $\mathcal{A}$
$T' \leftarrow Q - T$;
$Q' \leftarrow Q$;
$E' \leftarrow E$;
si $\mathcal{A}$ *n'est pas complet* **alors**
  Soit un nouvel état q_t;
  Créer un état final q_t;
  $Q' \leftarrow Q' \cup \{q_t\}$;
  $T' \leftarrow T' \cup \{q_t\}$;
  pour tous les $q \in Q', \sigma \in \Sigma$ **faire**
    si $\nexists\ q' \mid\ (q, \sigma, q') \in E'$ **alors**
      $E' \leftarrow E' \cup \{(q, \sigma, q_t)\}$;
    fin
  fin
fin
retourner $\mathcal{A}' = (Q', \Sigma, E', I, T')$

Algorithm 7: Création du complémentaire d'un automate fini déterministe

6.2.2 Concaténation

La concaténation de deux langages rationnels $\mathcal{L}_1$ et $\mathcal{L}_2$ consiste à former l'ensemble des mots m formés par concaténation des mots $m_1 \in \mathcal{L}_1$ et des mots $m_2 \in \mathcal{L}_2$. Plus formellement,

$$\mathcal{L}_{\text{concat}} = \{m_1 \cdot m_2 \mid m_1 \in \mathcal{L}_1 \text{ et } m_2 \in \mathcal{L}_2\}$$

Théorème 2 (Concaténation de langages rationnels) *Si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont des langages rationnels, alors $\mathcal{L}_{concat} = \mathcal{L}_1 \cdot \mathcal{L}_2$ est un langage rationnel.*

Preuve. Nous allons construire l'automate résultant de la concaténation de deux automates finis.
Soient $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$ deux automates finis. On suppose Q_1 et Q_2 disjoints[2]. On pose $I_1 = \{i_1\}$ et $I_2 = \{i_2\}$.

$\mathcal{A}_1 \cdot \mathcal{A}_2 = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, E, I_1, T)$ où

- $E = E_1 \cup E_2 \cup \{(q_1, \sigma, q_2) \mid q_1 \in T_1 \text{ et } (i_2, \sigma, q_2) \in E_2\}$;
- $T = T_1 \cup T_2$ si $i_2 \in T_2$, sinon $T = T_2$.

Supposons qu'il existe deux mots $m_1 \in \mathcal{L}_1$ et $m_2 \in \mathcal{L}_2$, par définition de l'automate $\mathcal{A}_1 \cdot \mathcal{A}_2$. Lire le mot m_1 mène à un état $q \in T_1$ puisque $E_1 \subset E$, puis toujours par construction, $\{(q_1, \sigma, q_2) \mid q_1 \in T_1 \text{ et } (i_2, \sigma, q_2) \in E_2\} \subset E$. Or, puisque $m_2 \in \mathcal{L}_2$, le premier symbole de ce mot mène forcément à un état q_2 depuis q. La suite du mot mène à un état final de T_2 et $T_2 \subseteq T$. Par conséquent, la concaténation des mots $m_1 \in \mathcal{L}_1$ et $m_2 \in \mathcal{L}_2$ est acceptée par l'automate $\mathcal{A}_1 \cdot \mathcal{A}_2$.
Supposons maintenant que l'un des deux mots ne soit pas dans l'un de ces deux langages. En suivant les ensembles de transitions, nous ne pourrons atteindre un état final. La concaténation de ces deux mots ne pourra donc pas être acceptée par l'automate $\mathcal{A}_1 \cdot \mathcal{A}_2$. Cet automate accepte donc exactement les mots formés par la concaténation des langages $\mathcal{L}_1$ et $\mathcal{L}_2$. □

[2]Sinon, il suffit de renuméroter les états de Q_2.

L'algorithme 8 réalisant la concaténation se fait donc pour un AFD en $O(|T_1| \times |\Sigma_2|)$ puisqu'il suffit de dupliquer les transitions partant de i_2, soit $|\Sigma_2|$ depuis chaque état de T_1. Pour un AFN, il est de l'ordre du nombre de transitions sortant de i_2 et du nombre d'états dans T_1.

Entrées: Deux automates $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$
Sorties: Un automate $\mathcal{A} = \mathcal{A}_1 \cdot \mathcal{A}_2$ acceptant la concaténation des langages $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$
$Q \leftarrow Q_1 \cup Q_2$;
$\Sigma \leftarrow \Sigma_1 \cup \Sigma_2$;
$E \leftarrow E_1 \cup E_2$;
pour tous les $q \in T_1, \sigma \in \Sigma_2$ **faire**
  si $\exists q_2 \in Q_2 \mid (i_2, \sigma, q_2) \in E_2$ **alors**
    $E \leftarrow E \cup (q, \sigma, q_2)$;
  fin
fin
$T \leftarrow T_2$;
si $i_2 \in T_2$ **alors**
  $T \leftarrow T \cup T_1$;
fin
retourner $\mathcal{A} = (Q, \Sigma, E, I_1, T)$

Algorithm 8: Création d'un automate représentant la concaténation de deux langages rationnels $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$

6.2.3 Union

L'union de deux langages rationnels $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage $\mathcal{L}$ qui accepte les mots de $\mathcal{L}_1$ et ceux de $\mathcal{L}_2$:

$$\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2 = \{m \in (\Sigma_1 \cup \Sigma_2)^* \mid m \in \mathcal{L}_1 \text{ ou } m \in \mathcal{L}_2\}.$$

Théorème 3 (Union de langages rationnels) *Si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont deux langages rationnels, alors $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ est un langage rationnel.*

Preuve. La preuve de cette propriété est basée sur un algorithme qui construit un automate reconnaissant $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$, étant donnés les deux automates $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$. Soit un nouvel état i. On pose $I = \{i\}$.
$\mathcal{A}_1 \cup \mathcal{A}_2 = (Q_1 \cup Q_2 \cup I, \Sigma_1 \cup \Sigma_2, E, I, T_1 \cup T_2)$, avec $E = E_1 \cup E_2 \cup \{(i, \sigma, q) | \exists (i_1, \sigma, q) \in E_1 \text{ ou } (i_2, \sigma, q) \in E_2\}$
La preuve que ce nouvel automate représente bien l'union d'$\mathcal{A}_1$ et $\mathcal{A}_2$ se fait de la même manière que pour la concaténation. □

L'algorithme 9 décrit la construction de l'union de deux automates.

Notons que cet algorithme produit un automate fini non déterministe si les deux automates possèdent deux transitions marquées par le même symbole au départ de leurs états initiaux respectifs. L'algorithme produit un automate possédant $|Q_1| + |Q_2|$ états. Sa complexité temporelle se porte uniquement sur la copie des transitions sortant des états initiaux, c'est-à-dire $O(|E_{i_1}| + |E_{i_2}|$.

6.2.4 Intersection

Nous avons précédemment utilisé l'intersection de deux langages rationnels pour modéliser sous forme d'automate deux règles de séquencement. Dans un problème de planification de personnel, chaque employé est soumis à un ensemble de règles. La plupart du temps ces règles doivent être toutes respectées. Si un

Entrées: Deux automates $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$
Sorties: Un automate $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ acceptant l'union des langages rationnels $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$
Soit i un nouvel état ;
$I \leftarrow \{i\}$;
$Q \leftarrow Q_1 \cup Q_2 \cup I$;
$T \leftarrow T_1 \cup T_2$;
$E \leftarrow E_1 \cup E_2$;
$\Sigma \leftarrow \Sigma_1 \cup \Sigma_2$;
Soit $E_{i_1} \subset E_1$ l'ensemble des transitions sortant de l'état i_1;
Soit $E_{i_2} \subset E_2$ l'ensemble des transitions sortant de l'état i_2;
pour tous les $(j, \sigma, q) \in E_{i_1} \cup E_{i_2}$ **faire**
  $E \leftarrow E \cup (i, \sigma, q)$;
fin
retourner $\mathcal{A} = (Q, \Sigma, E, I, T)$

Algorithm 9: Création d'un automate représentant l'union de deux langages rationnels $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$

automate représente une règle, l'intersection de tous les automates représente la conjonction de ces règles. C'est pourquoi l'intersection est l'opération que nous utilisons le plus.

L'intersection de deux langages rationnels $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage $\mathcal{L}$ qui accepte les mots communs à $\mathcal{L}_1$ et $\mathcal{L}_2$.

$$\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2 = \{m \in (\Sigma_1 \cap \Sigma_2)^* \mid m \in \mathcal{L}_1 \text{ et } m \in \mathcal{L}_2\}$$

Théorème 4 (Intersection de langages rationnels) *Si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont deux langages rationnels, alors $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$ est un langage rationnel.*

Preuve. La preuve de ce théorème se fait encore une fois par construction d'un automate acceptant les mots qui apparaissent dans les deux langages. Soient $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$ deux automates finis. Nous proposons de construire l'automate $\mathcal{A} = (Q, \Sigma, E, I, T) = \mathcal{A}_1 \cap \mathcal{A}_2$.
L'alphabet Σ est l'intersection des alphabets des automates d'origine, puisqu'un mot composé avec un symbole appartenant uniquement à un des alphabets ne pourra pas être accepté par l'autre automate. Les états Q de ce nouvel automate sont constitués de toutes les paires d'états (q_1, q_2) de l'ensemble $Q_1 \times Q_2$.
Une transition $((q_1, q_2), \sigma, (r_1, r_2))$ est dans E si et seulement si $(q_1, \sigma, r_1) \in E_1$ et $(q_2, \sigma, r_2) \in E_2$.
$I = I_1 \times I_2$ et $T = T_1 \times T_2$.
Supposons que les deux automates soient complets. Soit un mot m appartenant au langage $\mathcal{L}_1(\mathcal{A}_1)$ mais pas au langage $\mathcal{L}_2(\mathcal{A}_2)$.
Alors, lire le mot dans l'automate $\mathcal{A}_2$ mène à un état $q \notin T_2$. Par construction, lire ce mot dans $\mathcal{A}$ ne peut mener à un état final puisque $T = T_1 \times T_2$.
Au contraire, si le mot appartient aux deux langages, alors il existe un ensemble de transitions pour chaque automate menant à un état final. De par la construction de l'automate, ces transitions existent dans l'automate $\mathcal{A}$. □

L'algorithme 10 reproduit ce schéma. Sa complexité dépend du nombre d'états dans les automates dont on fait l'intersection ; mais aussi du nombre de transitions. L'opération $Q \leftarrow Q_1 \times Q_2$ créant tous les couples d'états a pour complexité $O(|Q_1| \times |Q_2|)$. La création des autres ensembles d'états, strictement inclus dans $Q_1 \times Q_2$ est donc dominée par cette opération. La création de l'ensemble des transitions n'est pas forcément dominée par la création des états puisqu'il peut y avoir plus de transitions que d'états. Or, dans l'algorithme que nous présentons, nous parcourons au moins $|E_1|$ transitions[3]. Par conséquent, la complexité de l'intersection de deux automates finis est $O(|Q_1| \times |Q_2| + |E_1|)$.

[3]Si $|E_2| < |E_1|$, il suffit d'inverser les transitions.

Entrées: Deux automates $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$
Sorties: Un automate $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$ acceptant l'intersection des langages rationnels $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$
$I \leftarrow I_1 \times I_2$;
$Q \leftarrow Q_1 \times Q_2$;
$T \leftarrow T_1 \times T_2$;
$E \leftarrow \emptyset$;
$\Sigma \leftarrow \Sigma_1 \cap \Sigma_2$;
pour tous les $(q, \sigma, q') \in E_1$ **faire**
  si $(q, \sigma, q') \in E_2$ **alors**
    $E \leftarrow E \cup (q, \sigma, q')$;
  fin
fin
retourner $\mathcal{A} = (Q, \Sigma, E, I, T)$

Algorithm 10: Création d'un automate représentant l'intersection de deux langages rationnels $\mathcal{L}(\mathcal{A}_1)$ et $\mathcal{L}(\mathcal{A}_2)$

6.2.5 Déterminisation

Un des aspects importants de la théorie des automates finis est la possibilité de transformer tout AFN en un AFD. Cette opération s'appelle la déterminisation. Elle est nécessaire dans de nombreux cas notamment pour le calcul du complémentaire d'un langage rationnel. En effet, l'algorithme présenté section 6.2.2 ne fonctionne que sur les AFD. L'algorithme des parties de Rabin-Scott permet de transformer un AFN en AFD. Comme expliqué dans le livre d'Hopcroft et al. [53], la complexité au pire cas est de $O(2^n)$ mais il est précisé que la plupart des langages rationnels définis sous forme d'un AFN n'engendre pas un AFD de taille exponentielle. Au cours de nos travaux, nous n'avons jamais rencontré un tel cas. Cependant, les algorithmes de modélisation automatique que nous avons présentés chapitre 5 utilisent intensivement la déterminisation. Il existe donc certainement des règles de séquencement qui provoqueraient une explosion de ces algorithmes.

6.2.6 Minimisation

La minimisation consiste à construire un AFD de taille minimale pour un langage rationnel donné. Les algorithmes de minimisation existants ne fonctionnent que sur des AFD. En effet, Hopcroft et al. [53] ont montré qu'il était impossible de trouver un algorithme polynomial pour la minimisation des AFN. Ainsi, lorsque nous parlons de minimiser un AFN, cela implique au préalable la déterminisation de cet automate.

Il existe trois algorithmes pour la minimisation :

- algorithme de Moore, de complexité $O(n^2|\Sigma|)$ mais ayant pour complexité moyenne $O(n \log n)$;
- algorithme d'Hopcroft, de complexité $O(n \log n)$;
- algorithme de Brzozowski, de complexité $O(2^n)$ basé sur l'algorithme des parties.

Dans la suite de cette thèse nous utiliserons toujours l'algorithme d'Hopcroft lorsque nous parlerons de minimisation. Ce choix a été réalisé lorsque nous testions la génération automatique de règles de séquencement. L'implémentation de l'algorithme d'Hopcroft nous a toujours donné les meilleurs résultats en temps de calcul.

6.2.7 Intersection ou union ?

Comme nous l'avons vu dans les sections précédentes, nous pouvons réaliser l'intersection, l'union d'automates, ainsi que calculer le complémentaire d'un automate, pour un alphabet donné. Dans le cadre des problèmes de planification de personnel, nous allons, la plupart du temps, réaliser l'intersection de nombreux automates afin d'agréger un maximum de règles. Nous avons vu dans le premier exemple de ce chapitre, que l'intersection peut réduire la taille de l'automate. Malheureusement cet argument ne s'applique pas dans la majorité des cas pratiques, où l'intersection oblige à expliciter de nombreuses transitions qui, par exemple, étaient représentées par une boucle dans l'un des automates.

L'intérêt d'agréger les règles en un seul automate est en réalité tout autre. Nous allons, en programmation par contraintes, être capable de faire des déductions à partir des automates modélisant ces règles. On comprend intuitivement qu'un automate agrégeant plusieurs règles permettra de faire plus de déductions, et donc de réduire l'espace de recherche. Le premier exemple du chapitre, illustre, au-delà du problème de la taille, le pouvoir de l'intersection. En effet, si nous devions générer un planning pour une personne sujette à ces deux règles, en regardant individuellement les automates, nous n'aurions pu faire aucune déduction. Une fois les règles agrégées, nous pouvons être certains que cet employé devra faire trois nuits en début d'horaire.

L'opération d'intersection n'est malheureusement pas gratuite, puisqu'elle produit pour deux automates dont les ensembles d'états seraient Q_1 et Q_2, $|Q_1| \times |Q_2|$ nouveaux états. Supposons maintenant que nous fassions l'intersection de 10 automates, possédant chacun 10 états. En appliquant l'algorithme d'intersection successivement aux automates, nous obtiendrions un automate, avant minimisation, possédant $10^{10} = 10000000000$ états. On comprend bien que nous ne serions pas en mesure de traiter un tel automate. Pourtant, l'intuition nous dit qu'il y a quelque chose de fortement illogique dans cet état. L'intersection de deux langages produit systématiquement un langage acceptant moins de mots que chacun des langages d'origine. Formellement, on écrit :

$$\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2 \implies (\mathcal{L} \subseteq \mathcal{L}_1) \wedge (\mathcal{L} \subseteq \mathcal{L}_2).$$

Si rien ne dit qu'un plus petit langage, dans le sens contenant moins de mots, doit avoir moins d'états, il paraît étrange que l'automate explose autant en taille. Si cela peut arriver en théorie, en pratique, après minimisation, l'intersection produit des automates drastiquement plus petits que ce que prévoit la théorie.

Une première solution à ce problème consiste à utiliser la minimisation après chaque intersection. Cela permet de réaliser l'intersection avec un automate que l'on sait minimal. Néanmoins, l'ordre dans lequel nous réalisons les intersections peut avoir un grand impact. La solution que nous préconisons est d'utiliser l'opération d'union plutôt que l'intersection. Comme nous l'avons vu section 6.2.3, la complexité de cet algorithme dépend uniquement du nombre de transitions sortant des états initiaux. Ainsi, l'union de nos 10 automates produira un automate ayant environ 100 états. Bien sûr, l'automate ne sera probablement pas déterministe. La question que l'on se pose maintenant est comment passer de l'intersection à l'union. Rappelons-nous que nos automates représentent des langages, donc des ensembles de mots. Les opérations d'union, de concaténation, d'intersection ou de complémentaire fonctionnent pour les automates finis. Or, nous savons que pour deux ensembles munis de telles opérations, on a :

$$A \cap B = \overline{\overline{A} \cup \overline{B}}.$$

De plus, nous rappelons que calculer le complémentaire d'un automate est quasiment gratuit. Nous

pensons donc que calculer le complémentaire de l'union des complémentaires sera une opération moins coûteuse que l'intersection simple des automates. Le lecteur consciencieux se posera la question suivante : et le déterminisme dans tout cela ? La question est légitime puisque lorsque nous minimiserons l'automate final résultant de l'union des complémentaires, l'algorithme de minimisation aura une complexité théorique qui dépendra également de la déterminisation. Or, cette opération peut produire $2^{|Q|}$ états. Cependant, nous pouvons montrer que la minimisation d'un automate fini non déterministe issu de l'union de deux automates finis déterministes $\mathcal{A}_1$ et $\mathcal{A}_2$ créera au maximum $|Q_1| \times |Q_2|$ états.

Lemme 1 *La minimisation d'un automate fini non deterministe $\mathcal{A} = (Q, \Sigma, E, I, T)$ issu de l'union de deux automate finis déterministes, $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$, produira un automate dont le nombre d'états est borné par $|Q_1| \times |Q_2|$.*

Preuve. La preuve de ce lemme résulte dans l'unicité de la forme minimale d'un automate fini déterministe. Ainsi, nous savons que dans le pire des cas, l'intersection de $\mathcal{A}_1$ et $\mathcal{A}_2$ produira un automate de taille $|Q_1| \times |Q_2|$ qui sera minimal. Or, en reprenant notre notation sur les complémentaires, on obtient $\mathcal{A}_1 \cap \mathcal{A}_2 = \sim (\sim \mathcal{A}_1 \cup \sim A_2)$.
Par conséquent, l'opération de complémentaire ne changeant pas le nombre d'états, la minimisation du terme droit de l'opération produira un automate dont le nombre d'états sera au plus $|Q_1| \times |Q_2|$. □

Ce que montre le lemme 1, est que dans le pire des cas, nous n'échapperons pas aux 10^{10} états lors de l'intersection de nos 10 automates. Mais nous avons pu observer qu'en pratique, cela permet de limiter la création d'états intermédiaires inutiles.

6.3 Intersection d'automates multi-pondérés

Nous avons, dans la section précédente 6.2, rappelé les opérations usuelles autorisées et bien définies pour les automates finis, représentant des langages rationnels. Ainsi, à travers deux exemples, nous avons montré l'utilisation de ces opérations pour agréger des règles de séquencement telles que celles définies dans le chapitre 5. Cela permet de modéliser un ensemble de règles, liées par des connecteurs logiques (ou,et,non) à l'aide d'un seul automate.

Afin de modéliser des règles de séquencement portant sur le nombre d'apparitions de certains motifs, nous avons introduit la notion d'automates pondérés. Associés à une variable compteur, ces automates permettent d'ajouter une contrainte sur la somme des transitions utilisées afin qu'un mot soit accepté. Pour agréger ces règles, nous devons définir une nouvelle opération sur les automates pondérés. L'intersection permet de construire un automate acceptant l'intersection des langages définis par ces automates pondérés.

Définition 31 *Soit deux automates finis déterministes multi-pondérés, $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, T_1)$ dans le semi-anneau $(\mathbb{Z}^{r_1}, \min, +)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, T_2)$ dans le semi-anneau $(\mathbb{Z}^{r_2}, \min, +)$. L'intersection $\mathcal{A}_1 \cap \mathcal{A}_2$ produit un automate $\mathcal{A}_{1\cap 2}$ dans le semi-anneau $(\mathbb{Z}^{r_1+r_2}, \min, +)$ tel que tout chemin dans cet automate existe dans $\mathcal{A}_1$ et $\mathcal{A}_2$, et a pour poids la concaténation des poids originaux.*

L'algorithme 11 permet de réaliser l'intersection de deux automates multi-pondérés. Il repose sur un marquage des transitions possédant un poids dépendant de l'état d'origine de la transition. Le marquage est effectué ainsi : si une transition possède une dimension de poids non nulle pour un symbole donné et si ce poids dépend de l'état d'origine de la transition, alors on ajoute une nouvelle transition non pondérée, mais par un symbole pris dans un ensemble autre que l'alphabet d'origine. On stocke l'information permettant de retrouver la dimension et la valeur du poids à partir du symbole. L'algorithme 12 permet de réaliser le marquage.

Une fois le marquage des deux automates effectués, nous réalisons une intersection des deux automates. L'idée est de réaliser une intersection classique en traitant à part les transitions marquées. En effet, si deux symboles de deux transitions marquées représentent le même symbole original, nous ajoutons ces deux transitions à l'automate. Une intersection classique aurait supprimé les transitions puisqu'elles n'ont pas le même symbole. L'algorithme 13 réalise cette opération.

Enfin, nous minimisons l'automate résultant et nous remplaçons dans les transitions les symboles marqués par leur valeur originale tout en ajoutant à la transition la dimension de poids représenté par le marquage. Si le symbole marqué vient du second automate $\mathcal{A}_2$, nous décalons la dimension du poids du nombre de dimension r_1 de $\mathcal{A}_1$. Les poids dépendant uniquement des symboles dans les automates originaux sont finalement replacés en utilisant le même décalage. C'est le sujet de l'algorithme 14.

Entrées: Deux automates multi-pondérés $\mathcal{A}_1$ et $\mathcal{A}_2$
Sorties: Un automate $\mathcal{A}$ intersection de $\mathcal{A}_1$ et $\mathcal{A}_2$
$\Sigma' \leftarrow \{marqueurs\}$;
$M_1 \leftarrow marque(\mathcal{A}_1, \Sigma')$;
$M_2 \leftarrow marque(\mathcal{A}_1, \Sigma')$;
$M \leftarrow intersectionMarque(M_1, M_2)$;
minimiser M;
$\mathcal{A} \leftarrow restaurer(M)$;
retourner $\mathcal{A}$

Algorithm 11: Intersection d'automates mutli-pondérés

Entrées: Un automate multi-pondéré $\mathcal{A} = (Q, \Sigma, E, I, T)$ et un alphabet $\Sigma' | \forall \sigma' \in \Sigma', \sigma' \notin \Sigma$
Sorties: Un automate M marqué
$M = (Q_m, \Sigma \cup \Sigma', E_m, I_m, T_m) \leftarrow \mathcal{A}$ sans les pondérations;
pour tous les $e = (q, \sigma, q', k) \in E$ **faire**
  si $\exists q \in Q, w(q, \sigma, \delta(q, \sigma)) \neq k$ **alors**
    pour tous les *dimensions de poids* $w \in k$ **faire**
      $E_m \leftarrow E_m - \{e\} + \{(q, marqueur_suivant(\Sigma'), q')\}$;
    fin
  fin
fin
retourner M

Algorithm 12: Marque : algorithme de marquage d'un automate multi-pondéré

6.4 Conclusion

Dans le cadre des problèmes de planification de personnel, nous avons, dans le chapitre 5, proposé une méthode systématique de modélisation des règles de séquencement sous forme d'automates simples ou pondérés. Afin d'unifier tous ces types de règles, nous avons présenté dans ce chapitre, l'utilisation que nous faisons des opérations usuelles sur les automates. De plus, afin de compléter ces opérations, nous avons défini l'intersection pour les automates pondérés dans un semi-anneau particulier. Lorsque toutes les règles ont été agrégées, l'automate résultant est un automate pondéré, possédant plusieurs poids sur chaque transition. Nous parlerons alors d'automates multi-pondérés.

Entrées: Deux automates marqués $M_1 = (Q_1, \Sigma, E_1, I_1, T_1)$ et $M_2 = (Q_2, \Sigma, E_2, I_2, T_2)$
Sorties: Un automate $M = (Q, \Sigma \cup \Sigma', E, I, T)$ marqué, intersection de M_1 etM_2
pour tous les $(q_1, q_2) \in Q_1 \times Q_2$ **faire**
  $Q \leftarrow Q \cup \{(q_1, q_2)\}$;
  si $q_1 \in I_1$ *et* $q_2 \in I_2$ **alors**
    $I \leftarrow (q_1, q_2)$;
  fin
  si $q_1 \in T_1$ *et* $q_2 \in T_2$ **alors**
    $T \leftarrow T \cup (q_1, q_2)$;
  fin
fin
pour tous les $(q_1, q_2) \in Q$ **faire**
  $Q_1^s \leftarrow \emptyset$;
  pour tous les *transitions* $e_2 = (q_2, \sigma_2, q_2')$ *sortants de* q_2 **faire**
    $Q_1^s \leftarrow Q_1^s \cup \delta(q_1, \sigma_2)$;
    $\Sigma'' \leftarrow \emptyset$;
    pour tous les *transitions* $e_1 = (q_1, \sigma_1, q_1')$ *sortants de* q_1 **faire**
      si σ_1 *et* σ_2 *ont le même symbole original* **alors**
        $Q_1^s \leftarrow Q_1^s \cup \{\delta(q_1, \sigma_1)\}$;
        $\Sigma'' \leftarrow \Sigma'' \cup \{\sigma_1\}$;
      fin
    fin
    pour tous les *états* $q_1' \in Q_1^s$ **faire**
      $E \leftarrow E \cup \{((q_1, q_2), \sigma_2, (q_1', q_2'))\}$;
      pour tous les *les symboles* $\sigma'' \in \Sigma''$ **faire**
        $E \leftarrow E \cup \{((q_1, q_2), \sigma'', (q_1', q_2'))\}$;
      fin
    fin
  fin
fin
retourner M

Algorithm 13: Intersection d'automates marqués

Entrées: Un automate marqué $M = (Q_m, \Sigma \cup \Sigma', E_m, I_m, T_m)$
Sorties: Un automate multi-pondéré $\mathcal{A} = (Q, \Sigma, E, I, T)$
$Q \leftarrow Q_m$;
$I \leftarrow I_m$;
$T \leftarrow T_m$;
pour tous les *transitions* $e = (q, \sigma_m, q') \in E_m$ **faire**
  si $\sigma_m \in \Sigma'$ **alors**
    $\sigma \leftarrow$ symbole original de σ_m;
    $r \leftarrow$ dimension originale du poids représenté par σ_m;
    $w \leftarrow$ poids original représenté par σ_l;
    si σ_m *originaire de* $\mathcal{A}_2$ **alors**
      $r \leftarrow r + r_1$;
    fin
    si $(q, \sigma, q', k) \in E$ **alors**
      $k[r] = w$;
    fin
    si $(q, \sigma, q', k) \notin E$ **alors**
      soit k un vecteur de dimension $r_1 + r_2$;
      $k[r] = w$;
      $E \leftarrow E \cup (q, \sigma, q', k)$;
    fin
  fin
fin
retourner $\mathcal{A}$

Algorithm 14: Restaurer : remet les pondérations sur les transitions d'un automate marqué

Chapitre 7

Contraintes pour automates multi-pondérés

Sommaire

DANS les chapitres précédents, nous avons présenté des algorithmes pour modéliser automatiquement des règles de séquencements issues de problèmes de planification de personnel sous forme d'automates finis et d'automates finis pondérés. Afin de modéliser des ensembles de règles nous avons présenté les opérations usuelles pour des automates finis. Nous avons également introduit un algorithme permettant l'intersection d'automates pondérés, afin de construire un automate multi-pondéré, représentant exactement l'ensemble des règles modélisées par les automates d'entrée.

Pour filtrer ces ensembles de règles deux approches s'offrent à nous. La première consiste à traiter indépendamment les règles non agrégées, à l'aide des contrainte `regular` et `cost-regular`. Néanmoins, la quantité de déductions, que peut faire une conjonction de telles contraintes, est limitée pour deux raisons :

- l'absence des informations déduites lors de l'agrégation des règles ;
- la non prise en compte du lien qui existe entre les différents coûts.

Dans ce chapitre, nous proposons, pour traiter ce problème, un nouvel algorithme de filtrage pour une contrainte automate multi-pondérée : `multicost-regular` [64]. Nous montrerons que cette contrainte permet de faire des déductions plus importantes qu'une simple conjonction de `cost-regular` ou qu'une

conjonction de `regular` et `global-cardinality`. La contrainte `soft-multicost-regular` sera par la suite définie, afin de gérer les fonctions de pénalisation des contraintes automates souples. Enfin, ces contraintes seront testées et comparées aux contraintes existantes.

7.1 multicost-regular

La contrainte `multicost-regular` fournit un algorithme de filtrage pour un automate multi-pondéré. Ses paramètres sont les suivants :

- $X = \{x_0, \ldots, x_{n-1}\}$, une séquence de n variable entières ;
- $Z = \{z_0, \ldots, z_r\}$, un vecteur de $r+1$ variables de bornes ;
- Π un automate multi-pondéré dans le semi-anneau $(\mathbb{N}^{r+1}, \min, +)$.

La contrainte `multicost-regular` est satisfaite, si et seulement si :

$$X \in \mathcal{L}(\Pi) \tag{7.1}$$
$$Z = w(X) \tag{7.2}$$

où $w(X)$ est le poids du mot formé par les valeurs de X.

Afin de faciliter la définition de la contrainte et de son algorithme de filtrage, nous considérerons Π comme un simple automate fini auquel nous associons une matrice de coûts d'affectation c. Ainsi, une transition $e \in E \subset (Q \times \Sigma \times Q \times \mathbb{N}^{r+1}) = (q_i, \sigma, q_j, N)$ sera décomposée en une transition $e = (q_i, \sigma, q_j)$ et la valeur de $c_{\sigma q_i}$ sera N.
Cette formulation permet de plus de spécifier un vecteur de coût différent pour une transition selon sa position dans le graphe support Π_n. Il suffit pour cela de rajouter une dimension à c.

De part sa sémantique et son algorithme de filtrage, la contrainte `multicost-regular` présente deux avantages majeurs :

- elle permet de modéliser de manière simple et concise un large panel de sous-problèmes que l'on retrouve notamment dans les problèmes de planification de personnel ;
- elle dispose d'un algorithme de filtrage efficace permettant de réaliser plus de filtrage que la conjonction des sous-contraintes modélisées par `multicost-regular`.

Dans cette section, nous expliciterons le problème sous-jacent à la contrainte : les problèmes de plus court/long chemin sous contraintes de ressources. Nous expliquerons ensuite notre choix du filtrage basé sur la relaxation lagrangienne, ainsi que son implantation dans le solveur de contraintes `CHOCO`. Nous conclurons la section par un ensemble d'évaluations des performances et des pouvoirs de modélisation de la contrainte `multicost-regular`.

7.1.1 RCSPP

La condition d'acceptation d'un mot par un automate multi-pondéré est multiple :

- le mot formé par la séquence des valeurs prises par les variables doit appartenir au langage de l'automate fini simple ;

- la somme de chaque vecteur de coûts d'affectation des valeurs doit être comprise entre deux vecteurs de bornes.

Si pour la contrainte `cost-regular` le problème sous-jacent est un plus court (resp. long) chemin dans le graphe support, le problème sous-jacent à la contrainte `multicost-regular` est un problème de plus court (resp. long) chemin sous contraintes de ressources (RCSPP, resp. RCLPP).

Le RCSPP (resp. RCLPP) a pour but de trouver le plus court (resp. plus long) chemin entre une source et un puits dans un graphe orienté multi-valué, de telle sorte que les quantités de ressources accumulées sur les arcs ne dépassent pas certaines limites. Même pour des graphes acycliques, ce problème est connu pour être NP-difficile [51].

Deux approches sont le plus souvent utilisées pour résoudre le RCSPP : la programmation dynamique et la relaxation lagrangienne. Les méthodes basées sur la programmation dynamique étendent les algorithmes usuels de plus court chemin par enregistrement des coûts sur chaque dimension à chaque nœud du graphe. De même que dans `cost-regular`, cela peut être adapté pour le filtrage en convertissant ces étiquettes de coûts en vecteurs de coûts mais la mémoire nécessaire pour l'algorithme serait trop importante car les principes de backtrack et de propagation utilisés en programmation par contraintes interdisent la suppression d'étiquettes par domination forte (i.e. tous les coûts d'une étiquette à un nœud sont plus grands que d'autres).

La première version implémentée de `multicost-regular` utilisait néanmoins ce principe et ne donnait plus de résultat à partir de 10 nœuds et 4 dimensions de coûts.

Nous avons par conséquent investigué l'approche par relaxation lagrangienne, qui peut aussi être adaptée au filtrage par l'algorithme `cost-regular` sans augmenter de manière drastique les besoins en mémoire.

La formulation mathématique de la contrainte nous permet d'identifier plus clairement le problème sous-jacent à une contrainte `multicost-regular`.

Relaxation lagrangienne du RCSPP. On considère un graphe orienté $G = (V, E)$ muni d'un nœud source s et d'un nœud puits t, et un ensemble de ressources $\mathcal{R}_1, ..., \mathcal{R}_R$. Pour tout $1 \leq r \leq R$, $\overline{z^r}$ (resp. $\underline{z^r}$) dénote la consommation maximum (resp. minimum[1]) de la ressource $\mathcal{R}_\nabla$ sur un chemin de s à t dans G, et c^r_{ij} est la consommation de la ressource $\mathcal{R}_\nabla$ sur tout arc $(i, j) \in E$. Le RCSPP se modélise par le programme linéaire en nombres binaires suivant :

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij} \qquad \text{s.t.} \tag{7.3}$$

$$\underline{z^r} \leq \sum_{(i,j)\in E} c^r_{ij} x_{ij} \leq \overline{z^r} \qquad \forall r \in [1..R] \tag{7.4}$$

$$\sum_{j\in V} x_{ij} - \sum_{j\in V} x_{ji} = \begin{cases} 1 & \text{if } i = s, \\ -1 & \text{if } i = t, \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i \in V \tag{7.5}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E. \tag{7.6}$$

Dans ce modèle, une variable de décision x_{ij} indique si l'arc (i, j) appartient à un chemin solution. Les contraintes (7.4) sont les contraintes de ressources et les contraintes (7.5) sont les contraintes de chemin.

Le principe de la relaxation lagrangienne repose sur le transfert de contraintes dites difficiles dans la fonction objectif où elles sont ajoutées avec un coût de violation $u \geq 0$ appelé *multiplicateur lagrangien.*

[1] Dans la définition originale du RCSPP, il n'y a pas de borne inférieure sur la capacité : $\underline{z^r}$

Le programme linéaire résultant, appelé *sous-problème lagrangien de paramètre u*, est une relaxation du problème original. Résoudre le *dual lagrangien* consiste à trouver les multiplicateurs $u \geq 0$ qui fournissent la meilleure relaxation, c'est-à-dire la plus grande borne inférieure possible.

Les contraintes difficiles du *RCSPP* sont les $2R$ contraintes de ressources (7.4). En effet, la suppression de ces contraintes produit un problème de plus court chemin qui se résout en temps polynomial. Soit P l'ensemble des solutions $x \in \{0,1\}^E$ satisfaisant les contraintes (7.5) : P est l'ensemble des chemins de s à t dans G. Le sous-problème lagrangien de paramètre $u = (u_-, u_+) \in \mathbb{R}_+^{2R}$ s'écrit :

$$SP(u) : f(u) = \min_{x \in P} cx + \sum_{r=1}^{R} u_+^r (c^r x - \overline{z^r}) - \sum_{r=1}^{R} u_-^r (c^r x - \underline{z^r}).$$

Une solution optimale x^u pour $SP(u)$ correspond ainsi à tout plus court chemin dans le graphe $G(u) = (V, E, c(u))$ défini par :

$$c(u) = c + \sum_{r=1}^{R} (u_+^r - u_-^r) c^r \tag{7.7}$$

et son coût est égal à :

$$f(u) = c(u)x^u + \kappa_u, \tag{7.8}$$
$$\text{avec } \kappa_u = \sum_{r=1}^{n} (u_-^r \underline{z^r} - u_+^r \overline{z^r}).$$

Résoudre le dual lagrangien. Le dual lagrangien consiste à trouver la meilleure borne inférieure $f(u)$, c'est-à-dire à maximiser la fonction concave linéaire par morceaux f :

$$LD : \quad f_{LD} = \max_{u \in \mathbb{R}_+^{2R}} f(u). \tag{7.9}$$

Plusieurs algorithmes permettent de résoudre le dual lagrangien. Nous nous sommes intéressés ici à l'algorithme du sous-gradient [88] pour sa simplicité d'utilisation et parce qu'il ne nécessite pas l'utilisation d'un solveur linéaire.

L'algorithme du sous-gradient résout de manière itérative, pour différentes valeurs de u, le sous-problème $SP(u)$. À chaque itération, le vecteur u est mis à jour suivant la direction d'un super-gradient Γ de f avec un pas de longueur μ : $u^{p+1} = max\{u^p + \mu_p \Gamma(u^p), 0\}$.

Il existe plusieurs manières de choisir la longueur de pas afin de garantir la convergence vers f_{LD} de l'algorithme du sous-gradient (voir par exemple [24]).

Notre implémentation repose sur une longueur de pas standard $\mu_p = \mu_0 \epsilon^p$ avec μ_0 et $\epsilon < 1$ choisis « suffisamment » grands (nous avons choisi empiriquement $\mu_0 = 10$ et $\epsilon = 0.8$). Le super-gradient $\Gamma(u)$ est calculé à partir de la solution optimale $x^u \in P$ retournée par la résolution de $SP(u)$: $\Gamma(u) = ((c^r x^u - \overline{z^r})_{r \in [1..R]}, (\underline{z^r} - c^r x^u)_{r \in [1..R]})$.

7.1.2 Filtrage de `multicost-regular` basé sur la relaxation lagrangienne

Sellmann [87] a établi le principe de l'utilisation de la relaxation lagrangienne d'un programme linéaire pour filtrer des contraintes d'optimisation. Nous appliquons ce principe au RCSPP/RCLPP pour le filtrage de `multicost-regular`. L'algorithme résultant est un simple procédé itératif dans lequel le filtrage est

pris en charge par cost-regular dans Π_n pour différentes fonctions de coûts agrégées. Ainsi, comme montré dans [87], si une valeur est inconsistante dans au moins un des sous-problèmes lagrangiens, alors elle l'est également pour le problème original. C'est la clé du filtrage par relaxation lagrangienne.

Théorème 5 *(i) Soit P un programme linéaire de valeur minimale $f^* \leq +\infty$, soit $\overline{z} \leq +\infty$ une borne supérieure de f^*, et soit $SP(u)$ un sous-problème lagrangien de P de valeur minimale $f(u) \leq +\infty$. Si $f(u) > \overline{z}$ alors $f^* > \overline{z}$.*
(ii) Soit x une variable de P et v une valeur de son domaine. Soit $P_{x=v}$ (resp. $SP(u)_{x=v}$) la restriction de P (resp. $SP(u)$) à l'ensemble des solutions telles que $x = v$, et soit $f^*_{x=v} \leq +\infty$ (resp. $f(u)_{x=v} \leq +\infty$) sa valeur minimale. Si $f(u)_{x=v} > \overline{z}$ alors $f^*_{x=v} > \overline{z}$.

Preuve. La proposition (i) du théorème 5 est triviale car, $SP(u)$ étant une relaxation de P, $f(u) \leq f^*$. La proposition (ii) se déduit de (i) et le fait d'ajouter une contrainte $x = v$ à P puis d'appliquer une relaxation lagrangienne ou d'appliquer la relaxation lagrangienne à P puis ajouter la contrainte $x = v$ produit la même formulation. □

On établit la relation entre une instance de multicost-regular(X, Z, Π, c), avec $|Z| = R+1$, et deux instances du RCSPP et du RCLPP comme suit : on sélectionne une variable de coût, par exemple z^0, et on crée R ressources, une par variable de coûts restante. Le graphe considéré est $G = (\Pi_n, c^0)$. Une solution réalisable du RCSPP (resp. RCLPP) est un chemin dans Π_n depuis la source (dans la couche 0) vers un puits (dans la couche n) dont la consommation pour chaque ressource $1 \leq r \leq R$ est au moins $\underline{z^r}$ et au plus $\overline{z^r}$. De plus, nous voulons limiter par une borne supérieure $\overline{z^0}$ la valeur minimale du RCSPP (resp. une borne inférieure $\underline{z^0}$ la valeur maximale du RCLPP). Les arcs de Π_n coïncident avec les variables binaires du modèle linéaire de ces deux instances.

Intéressons-nous à un sous-problème lagrangien $SP(u)$ du RCSPP (l'approche est symétrique pour le RCLPP). Une légère modification de l'algorithme de cost-regular permet de résoudre $SP(u)$, mais aussi de filtrer des arcs de Π_n selon le théorème 5 et de mettre à jour la borne inférieure $\underline{z^0}$. L'algorithme considère d'abord $c^0(u)$, défini par l'équation (7.7), comme valuation du graphe Π_n. Puis, il calcule pour chaque nœud (i, q), le plus court chemin $\underline{k^-_{iq}}$ depuis la couche 0 et le plus court chemin $\underline{k^+_{iq}}$ vers la couche n. On obtient la valeur optimale $f(u) = \underline{k^+_{0s}} + \kappa_u$. Comme il s'agit d'une borne inférieure pour z^0, on peut éventuellement mettre à jour $\underline{y^0} = \max\{f(u), \underline{z^0}\}$. Les arcs sont ensuite à nouveau parcourus, afin de filtrer tout arc $((i-1, q), a, (i, q')) \in \delta_i$ tel que $\underline{k^-_{(i-1)q}} + c^0(u)_{ia} + \underline{k^+_{iq'}} > \overline{z^0} - \kappa_u$.

L'algorithme de filtrage complet de multicost-regular procède comme suit : u est initialisé à 0. L'algorithme du sous-gradient guide le choix des sous-problèmes lagrangiens auxquels est appliqué l'algorithme de filtrage décrit ci-dessus. Dans nos expérimentations, le nombre d'itérations de l'algorithme du sous-gradient est limité à 20 (il termine le plus souvent en 5 ou 6 itérations). On résout d'abord le problème de minimisation (RCSPP) puis celui de maximisation (RCLPP). Enfin, l'algorithme original de cost-regular est utilisé sur chaque variable de coûts de manière indépendante afin de réduire leurs bornes (il peut également filtrer certains arcs, même si cela n'est jamais arrivé au cours de nos tests).

7.1.3 Évaluations

Pour évaluer l'efficacité du filtrage de la contrainte multicost-regular nous avons utilisé les instances du problème de planification benchmark introduit par Demassey et al [43]. Nous rappelons que les règles de cette instance sont issues d'un problème réel de planification d'horaire d'un magasin de vente au détail.

Modélisation

Une journée de 24 heures est découpée en 96 périodes de 15 minutes. Nous différencions les activités travaillées $W = A, B, C, \ldots$ d'activités spéciales (P une pause, O le repos au début et à la fin de la journée et L le repas). On note $S = W \cup \{P, O, L\}$.

Les règles s'appliquant sur le planning d'une journée pour une personne sont les suivantes :

1. La journée hors repos O dure entre 3 et 8 heures ;
2. Si une activité $w \in W$ est effectuée, alors elle dure au moins 1 heure ;
3. Une pause ou un repas est obligatoire après une activité travaillée ;
4. Au maximum un repas et deux pauses par jour ;
5. Une pause dure au maximum 15 minutes ;
6. Le repos se situe en début ou en fin de journée ;
7. Une pause est obligatoire par jour.

Afin de mesurer uniquement la performance de la contrainte, nous n'avons résolu le problème que pour un employé. Ce problème ne pouvant alors être modélisé qu'à l'aide d'une seule contrainte `multicost-regular`. Les règles $2, 3, 5$ et 6 sont modélisées à l'aide d'un automate fini Π_s. Les autres seront modélisées à l'aide de compteurs d'activités dans la contrainte `multicost-regular`.

Modèle basé sur `multicost-regular` :

- X une séquence de 96 variables à valeurs dans S ;
- Π_s l'automate fini représentant les règles $2, 3, 5$ et 6
- z_0 une variable de borne de domaine $\mathbb{N}^+$;
- z_1 une variable de borne de domaine $[\![0, 4]\!]$;
- z_2 une variable de borne de domaine $[\![1, 2]\!]$;
- z_3 une variable de borne de domaine $[\![12, 32]\!]$;
- C une matrice de coût d'affectation de dimension $|S| \times 4$.

La variable z_0 représente le coût d'une solution. z_1 sera le compteur associé aux repas, z_2 le compteur associé aux pauses et z_3 le compteur d'activités travaillées.

La matrice C est alors définie ainsi :

$$\forall s \in S, \forall r \in [\![0, 3]\!], C_{sr} = \begin{cases} rand() & \text{si } r = 0, \\ 1 & \text{si } s = L \text{ et } r = 1, \\ 1 & \text{si } s = P \text{ et } r = 2, \\ 1 & \text{si } s \in W \text{ et } r = 3, \\ 0 & \text{sinon.} \end{cases}$$

La seule contrainte utilisée est alors `multicost-regular`(X, Z, Π_s, C).

Nous générons aléatoirement des coûts d'affectation et nous faisons varier le nombre d'activités réalisables afin de mesurer le passage à l'échelle de la contrainte lorsque la taille de l'automate augmente. Pour

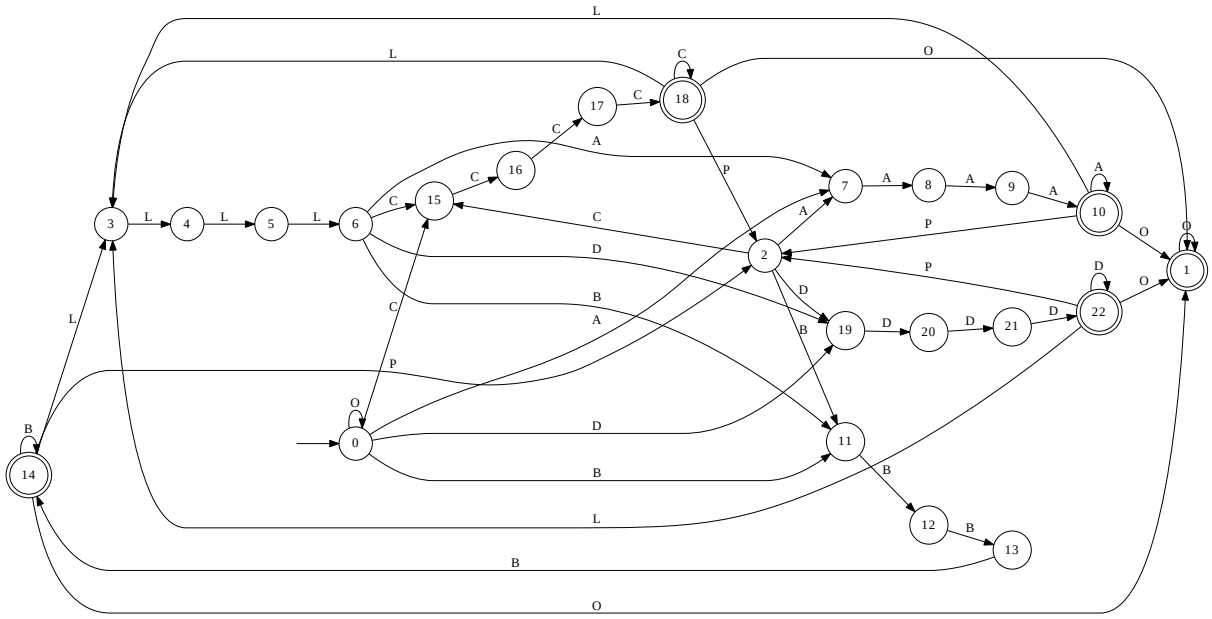


FIGURE 7.1 – Automate représentant l'ensemble des règles de séquencement pour 4 activités

chaque taille d'automate, nous avons généré 20 matrices de coûts d'affectation des activités différentes. La figure 7.1 montre l'automate construit pour 4 activités (A, B, C, D).

Les résultats obtenus sont comparés tout d'abord avec un modèle similaire basé sur des conjonctions de contraintes `cost-regular`. Ici, pour chaque dimension de coût r, nous construisons une contrainte `cost-regular`(X, Π_s, C_r).

Les dimensions de coûts 1 à 3 permettent de compter le nombre d'occurrences de certaines valeurs. Ainsi nous pouvons également comparer notre modèle `multicost-regular` avec une conjonction des contraintes `cost-regular` et `global-cardinality`. La contrainte `cost-regular` est posée sur la première dimension d'indice 0. Nous introduisons une nouvelle séquence de variables Y de domaines $\{L, P, O, T\}$. Ces variables sont liées à X de la manière suivante :

$$\forall i \in [\![1, |X|]\!], y_i = \begin{cases} x_i & \text{si } x_i \in \{L, P, O\}, \\ 3 & \text{sinon.} \end{cases}$$

Ainsi liées par cette contrainte, les variables Y permettent de connaître le nombre de quarts travaillés en comptant le nombre de variables ayant pour valeur 3.

Nous posons ensuite la contrainte `global-cardinality`$(Y, [0, 1, 64, 12], [4, 2, 84, 32])$ qui va s'assurer que le nombre d'apparitions de L sera compris entre 0 et 4, P entre 1 et 2, O entre 64 et 84 (entre 16 et 21 heures de repos) et T entre 12 et 32.

Notons que du point de vue de la modélisation, le modèle `multicost-regular` est plus direct et évite le besoin de contraintes de channelling.

Résultats

Les résultats de nos expérimentations sont présentés dans le tableau 7.1. Nous avons comparé les modèles avec des automates de taille croissante, pour une activité, l'automate possède 11 états et 15 transitions, tandis que pour 50 activités il y a 207 états et 505 transitions. Le temps de résolution

maximal pour une instance a été fixé arbitrairement à 10 minutes. Les heuristiques de choix de valeurs et variables sont les heuristiques par défaut du solveur de contraintes `CHOCO`.

On remarque que dans tous les cas, le modèle `multicost-regular` produit des résultats optimaux plus rapidement et surtout en un nombre de nœuds de 6 à 700 fois plus petit. Cela montre que le filtrage basé sur la relaxation lagrangienne dans `multicost-regular` est plus efficace qu'une conjonction de contraintes `cost-regular` ou d'une `cost-regular` et d'une `global-cardinality`.

Toutes les instances sont résolues dans un temps moyen inférieur à 5 secondes pour le modèle `multicost-regular`. Si le temps augmente légèrement lorsque la taille de l'automate augmente, le nombre de nœuds nécessaires à la résolution reste stable. Au contraire, pour les deux autres modèles, le nombre de nœuds augmente de manière dramatique lorsque la taille de l'automate augmente. Cela impacte grandement le temps de résolution qui dépasse rapidement la limite des 10 minutes à partir de 15 activités. Par curiosité, nous avons résolu entièrement une instance de 50 activités avec le modèle basé sur une conjonction de `cost-regular`. La preuve de l'optimalité a été obtenue en 5 heures.

Grâce au filtrage efficace de `multicost-regular`, le temps entre la découverte de la meilleure solution et de la preuve d'optimalité est très faible. Par opposition, on remarque que même si la meilleure solution a été trouvée relativement rapidement lorsque l'on utilise les autres modèles, la preuve d'optimalité nécessite un temps beaucoup plus grand. Ceci est une nouvelle preuve du filtrage supérieur de `multicost-regular`.

	Modèle CCR				Modèle MCR			
n	résolus	preuve(s)	meilleure(s)	#nœuds	résolus	preuve(s)	meilleure(s)	#nœuds
1	100%	1.2	1.0	3654	100%	0.0	0.0	41
2	100%	2.1	0.9	1563	100%	0.1	0.1	68
4	100%	13.9	8.8	6401	100%	0.2	0.1	67
8	100%	101.7	49.8	19637	100%	0.3	0.2	52
10	100%	297.2	167.8	44530	100%	0.4	0.4	63
15	50%	353.6	135.2	28699	100%	0.8	0.7	63
20	10%	144.3	10.0	7256	100%	1.2	1.0	64
30	10%	396.8	67.6	12219	100%	1.8	1.5	62
50	0%				100%	5.0	4.8	65
	Modèle CR + GCC				Modèle MCR			
n	résolus	preuve(s)	meilleure(s)	#nœuds	résolus	preuve(s)	meilleure(s)	#nœuds
1	100%	0.3	0.2	225	100%	0.0	0.0	41
2	100%	0.6	0.3	393	100%	0.1	0.1	68
4	100%	2.9	2.3	1199	100%	0.2	0.1	67
8	100%	17.9	13.2	3597	100%	0.3	0.2	52
10	100%	50.0	47.7	7615	100%	0.4	0.4	63
15	100%	58.1	47.1	6233	100%	0.8	0.7	63
20	100%	58.1	44.0	4577	100%	1.2	1.0	64
30	80%	153.1	127.4	6080	100%	1.8	1.5	62
50	40%	460.0	65.6	6747	100%	5.0	4.8	65

TABLE 7.1 – Comparaison du modèle `multicost-regular` aux modèles usuels

7.2 Soft-multicost-regular

La contrainte globale `multicost-regular` permet de modéliser des règles de séquencement complexes à l'aide d'un automate multi-pondéré. Nous venons de voir à travers le dernier exemple section 7.1.3 qu'elle offre un filtrage beaucoup plus efficace qu'une conjonction de `cost-regular` ou d'une `cost-regular` avec

une `global-cardinality`. Néanmoins l'objectif étant de résoudre des problèmes de planification de personnel, il faut noter qu'une grande partie de ces derniers sont sur-contraints. Nous proposons pour cela une adaptation de la contrainte globale `multicost-regular`. La contrainte `soft-multicost-regular` permet de pénaliser la violation des variables compteurs de `multicost-regular`.

La signature de la contrainte est la suivante : `soft-multicost-regular`(X, Y, Z, v, P, Π) où :

- $X = \{x_0, \ldots, x_{n-1}\}$ est une séquence de variables entières ;
- $Y = \{y_0, \ldots, y_r\}$ est un ensemble de variables de bornes, équivalent aux compteurs dans `multicost-regular` ;
- $Z = \{z_0, \ldots, z_r\}$ est un ensemble de variables de coûts de violation ;
- v une variable de borne représentant le coût de violation global de la contrainte ;
- $P = \{p_0, \ldots, p_r\}$ un ensemble de fonctions de violation liant Y et Z ;
- Π un automate multi-pondéré dans le semi-anneau $(\mathbb{N}^r, \min, +)$.

La contrainte `soft-multicost-regular`(X, Y, Z, v, P, Π) est satisfaite si et seulement si :

$$X \in \mathcal{L}(\Pi) \tag{7.10}$$

$$v = \sum_r z_r \tag{7.11}$$

$$z_r = p_r(y_r) \qquad \forall r \tag{7.12}$$

$$Y = w(X) \tag{7.13}$$

où $w(X)$ est le poids du mot formé par les valeurs de X.

7.2.1 Modéliser les coûts de violation.

Avant de parler du filtrage d'une telle contrainte, intéressons-nous à ce qu'apporte `soft-multicost-regular` pour modéliser les coûts de violation. Une méthode consiste à donner des bornes plus larges aux variables compteurs y_r tout en pénalisant l'utilisation des valeurs au-delà des bornes préférentielles. Cette pénalisation est modélisée à l'aide d'une variable de coûts z_r et d'une table de paires autorisées définissant en extension une fonction p_r de violation. Cette fonction définit la relation suivante :

$$\forall r, \quad z_r = p_r(y_r).$$

Par exemple, considérons la règle indiquant que le motif NN doit apparaître entre 1 et 2 fois. Si cette règle est obligatoire, la définition d'une variable compteur $y_r \in [1, 2]$ ainsi que l'intégration de l'automate définissant ce motif dans la `multicost-regular` sont suffisantes. Supposons maintenant que le nombre d'apparitions de ce motif puisse être violé et que le coût de violation pour chaque déviation augmente de manière quadratique ($p_r(y_r) = c_r.y_r^2$) avec, par exemple, un coût de 10 pour une déviation aux bornes de 1 ($1^2 * 10$), de 40 pour une déviation de 2 ($2^2 * 10$), etc. Il suffit alors d'étendre les bornes de y_r de 0 au nombre maximum de fois que le motif peut apparaître jusqu'à l'horizon du planning. Une table de relation entre z_r et y_r est alors calculée à partir de la fonction de violation. Pour un horizon de planning de 6 jours, le motif NN peut apparaître entre 0 et 5 fois. Le tableau 7.2 présente la liste des paires autorisées entre z_r et y_r.

y_r	0	1	2	3	4	5
z_r	10	0	0	10	40	90

TABLE 7.2 – Paires autorisées entre une variable compteur y_r et son coût de violation z_r

Avec cette modélisation, nous pouvons définir des fonctions de coûts de violation non linéaires et associer un coût de violation z à chaque `multicost-regular` comprenant un ensemble S de compteurs souples, défini par $v = \sum_{r \in S} z_r$.

7.2.2 Filtrage du coût de violation global.

L'efficacité de la contrainte `multicost-regular` repose sur l'agrégation des différentes dimensions de coûts et compteurs au sein d'un algorithme de filtrage basé sur la relaxation lagrangienne [64]. La modélisation des coûts de violation proposée ci-dessus, extérieure à la contrainte, ne permet pas d'utiliser la structure de graphe sous-jacente à `multicost-regular` pour calculer de bonnes bornes pour la variable de coût global v. Nous proposons donc d'adapter la `multicost-regular` à ce cas : `soft-multicost-regular` intègre la variable v dans la contrainte afin de calculer de meilleures bornes. Rappelons que le filtrage de `multicost-regular` s'effectue sur le graphe acyclique $\Pi_n = G(Q, E)$, qui est l'automate Π acceptant uniquement les mots de longueur n.

7.2.3 Filtrage pour `soft-multicost-regular`

Nous allons tenter de réaliser un filtrage par relaxation lagrangienne pour la contrainte `soft-multicost-regular`. Pour faciliter l'écriture mathématique de la relaxation lagrangienne, nous introduisons les notations suivantes :

- $\{c_{er}\}_{er}$ est le coût d'utilisation de l'arc $e \in \Pi_n$ pour un compteur r issu de l'automate multi-pondéré Π ;
- $(\delta_e^x)_{e \in E} \in \{0,1\}^E$ le vecteur d'incidence associé au chemin X dans Π_n ;
- Γ l'ensemble des chemins dans Π_n.

La cohérence des contraintes (7.11) et (7.12) est maintenue par des contraintes de somme et de table. Ces dernières sont intégrées au filtrage de la `soft-multicost-regular` de manière à éviter les pertes de performances dues aux mécanismes internes du solveur.

Calculer les bornes de v au sein de `soft-multicost-regular` revient à résoudre les problèmes suivants :

$$\begin{aligned} \underline{v} = \min & \sum_r p_r(y_r) \\ \text{s.t. } y_r = & \sum_{(e) \in \Pi_n} c_{er} \delta_e^x \qquad \forall r \\ & \delta_e^x \in \Gamma \end{aligned}$$

et

$$\begin{aligned} \overline{v} = \max & \sum_r p_r(y_r) \\ \text{s.t. } y_r = & \sum_{(e)\in\Pi_n} c_{er}\delta_e^x \qquad \forall r. \\ & \delta_e^x \in \Gamma \end{aligned}$$

Par souci de concision, nous ne considérons maintenant que le calcul de $\underline{z}$, le calcul de $\overline{z}$ étant symétrique. Ce problème d'optimisation sous-jacent à la `soft-multicost-regular` possède la même structure que celui présenté dans la `multicost-regular`. Le filtrage basé sur la relaxation lagrangienne s'adapte de telle façon qu'un nouveau sous-problème lagrangien est défini pour tout $\lambda \in \mathbb{R}^r$:

$$\begin{aligned} SP(\lambda): \quad g(\lambda) = & \min \sum_r p_r(y_r) - \lambda_r y_r \\ & + \min_{\delta_e^x \in \Gamma} \sum_{e\in\Pi_n} \delta_e^x \sum_r \lambda_r c_{er} \end{aligned}$$

Pour tout vecteur λ, résoudre $SP(\lambda)$ revient à résoudre $r+1$ problèmes indépendants.

Pour tout r on cherchera $y_r \in \mathbb{Z}_+$ qui minimise $\hat{g}_r(\lambda) = p_r(y_r) - \lambda_r y_r$.

La mesure de pénalité p_r n'étant pas forcément monotone, il faut calculer $\hat{g}_r(\lambda)$ pour toutes les valeurs y_r. y_r étant un compteur d'occurrences cela ne concernera jamais plus que quelques dizaines de valeurs en pratique. Notons que, plus l'on connaîtra les propriétés de p_r, plus on pourra optimiser ce calcul.

On cherche également à minimiser

$$\sum_{e\in\Pi_n} \delta_e^x \sum_r \lambda_r c_{er}.$$

Il s'agit ici de trouver le plus court chemin dans le graphe Π_n dont le poids des arcs e est $\sum_r \lambda_r c_{er}$.

Résoudre le dual lagrangien, revient à trouver le vecteur $\lambda*$ qui maximise g. Cela peut être fait en utilisant une implémentation de la méthode du sous-gradient telle celle décrite dans [64].

Cette modification de l'algorithme de la `multicost-regular` permet donc d'agréger les coûts de violation et d'affectation en un compteur global tout en fournissant des bornes de bonne qualité à ce compteur. Au sein d'un problème de conception d'horaires, l'usage de cette variable est double : accéder rapidement à une bonne estimation du coût de violation global pour un employé et être capable d'imposer une borne sur ce coût de violation.

Notons néanmoins que lorsqu'une fonction de pénalisation n'est pas convexe, l'algorithme du sous-gradient ne converge plus forcément. Cela signifie que nous ne trouverons pas forcément les meilleures bornes pour v. Cependant, chaque sous-problème étant une relaxation, les valeurs qu'il permet de filtrer sont valides dans le problème maître et les bornes de v peuvent être également améliorées. Le filtrage restera tout de même meilleur avec des fonctions de pénalisation convexes.

7.2.4 Évaluations

Afin d'évaluer la contrainte `soft-multicost-regular` nous avons établi le protocole suivant : pour chaque instance de la section 7.1.3 résolue pour évaluer `multicost-regular`, nous avons imposé que la variable de coût soit inférieure à l'optimale trouvée. A ce stade, il n'y a pas de solution. Nous relâchons alors les contraintes concernant les activités et les pauses. Plutôt qu'une ou deux pauses par jour, nous

autorisons entre 0 et 4 pauses par jour. De même, nous autorisons entre 2 et 10 heures de travail par jour plutôt qu'entre 3 et 8. Nous choisissons de pénaliser par un coût de 10 chaque quart d'heure en plus ou en moins ajouté aux bornes initiales. Ainsi, travailler 10 heures dans une journée ajoutera un coût de $2 * 4 * 10 = 80$. L'objectif du problème devient alors de minimiser le coût de violation global.

Nous proposons deux modèles pour résoudre ce problème, l'un basé sur la contrainte `multicost-regular` et l'autre sur `soft-multicost-regular`.

Base commune

- X une séquence de 96 variables à valeurs dans S ;
- Π_s l'automate fini représentant les règles $2, 3, 5$ et 6 ;
- y_0 une variable de borne de domaine $\mathbb{N}^+$;
- y_1 une variable de borne de domaine $[\![0, 4]\!]$;
- y_2 une variable de borne de domaine $[\![0, 4]\!]$;
- y_3 une variable de borne de domaine $[\![8, 40]\!]$
- z_2 une variable entière de domaine $\{0, 10, 20\}$;
- z_3 une variable entière de domaine $\{0, 10, 20, 30, 40, 50, 60, 70, 80\}$;
- v une variable de borne de domaine $[\![0, 100]\!]$;
- C une matrice de coût d'affectation de dimension $|S| \times 4$.

La variable y_0 représente le coût d'une solution tandis que les variables y_1, y_2 et y_3 représentent les compteurs associés aux repas, pauses et activités travaillées. z_2 et z_3 sont les variables de violations des règles sur le nombre de pauses et d'activités travaillées. v est la variable de violation totale. X, Π_s et C sont définis de la même manière que pour le problème de la section 7.1.3.

Modèle basé sur `multicost-regular`

Pour modéliser la violation de la contrainte, nous devons lier les variables compteurs y_2 et y_3 aux variables de violation z_2 et z_3.

Nous utilisons pour cela deux contraintes en extension définissant l'ensemble des couples autorisés. Pour `feasPairAC`(y_2, z_2), la liste des couples autorisés est la suivante : $\{(0, 10), (1, 0), (2, 0), (3, 10), (4, 20)\}$. Si une solution comporte 4 pauses, alors la variable z_2 vaudra 20. La liste des couples autorisés pour la contrainte `feasPairAC`(y_3, z_3) est construite de la même manière.

Afin de connaître la violation globale d'une solution, nous posons $v = z_2 + z_3$.

Finalement la contrainte `multicost-regular`(X, Y, Π_s, C) est ajoutée pour modéliser les contraintes de séquencement définies par l'automate et maintenir les compteurs Y.

Modèle basé sur `soft-multicost-regular`

Contrairement au modèle précédent, les contraintes liant les variables compteurs et de violation sont intégrées à la contrainte `soft-multicost-regular`. Cependant, il faut définir les fonctions de pénalisation les liant.

Soit $p : \mathbb{N}^5 \rightarrow \mathbb{N}$ tel que

$$p(minP, minV, maxP, maxV, val) = \begin{cases} (minP - val) * minV & \text{si } val < minP, \\ (val - maxP) * maxV & \text{si } val > maxP, \\ 0 & \text{sinon.} \end{cases}$$

où $minP$ (resp. $maxP$) désigne la borne inférieure (resp. supérieure) préférentielle et $minV$ (resp. $maxV$) le coût de violation pour chaque déviation de la borne inférieure (resp. supérieure) préférentielle.

Pour résoudre le problème il suffit alors de poser la contrainte `soft-multicost-regular`$(X, Y, Z, v, P, \Pi_s, C)$, avec $P = [p(1, 10, 2, 10, y_2); p(12, 10, 32, 10, y_3)]$.

Pour les deux modèles, il suffit alors d'utiliser le solveur de contraintes pour minimiser la variable v.

Résultats

Nous avons réalisé cette comparaison sur les mêmes instances que précédemment en suivant le protocole défini en début de section. Nous ne nous sommes intéressés qu'aux plus grandes instances avec $20, 30$ et 50 activités.

Les résultats présentés dans le tableau 7.3 ne produisent pas le résultat attendu. En effet, si les temps de calculs sont comparables, voire un peu meilleurs avec le modèle basé sur la contrainte `soft-multicost-regular`, le nombre de nœuds pour trouver la violation minimale est supérieur. L'analyse de l'algorithme de filtrage nous a montré que les sous-problèmes lagrangiens générés par la `soft-multicost-regular` sont de moins bonne qualité (i.e. ils permettent moins de filtrage) que ceux générés pas la contrainte `multicost-regular`. L'impact positif de `soft-multicost-regular` sur le temps est en réalité dû à l'absence de mécanisme de communication entre les différentes contraintes, puisqu'elles sont toutes intégrées dans le second modèle. D'autres expérimentations sur des problèmes plus complexes tels ceux traités au chapitre 10 nous ont permis de faire les mêmes conclusions. Néanmoins, il faut noter que l'apport en terme de modélisation de la contrainte `soft-multicost-regular` n'est pas négligeable puisqu'elle permet d'intégrer facilement des fonctions de pénalisation complexe au sein d'une contrainte automate multi-pondérée. La forme de ces fonctions de pénalisation influençant la génération des sous-problèmes lagrangiens, nous pensons pouvoir améliorer encore le filtrage de cette contrainte.

	Modèle MCR		Modèle SMCR	
n	preuve (s)	#nœuds	preuve (s)	#nœuds
20	1.4	43	1.4	44
30	2.6	51	2.2	63
50	6.1	50	6.3	62

TABLE 7.3 – Comparaison des modèles `multicost-regular` et `soft-multicost-regular`

7.3 Conclusion

Lorsque l'on conçoit un logiciel d'aide à la décision, la modélisation du problème que l'on cherche à résoudre est une étape critique et difficile. C'est pourquoi, nous nous sommes attachés, dans la deuxième partie de cette thèse, à proposer des techniques de modélisation automatique pour un grand nombre de règles de séquencement, et avons proposé des algorithmes efficaces pour résoudre ces modèles.

Pour chaque automate représentant une règle de séquencement, on peut associer une contrainte `regular` ou `cost-regular`. Néanmoins, la quantité de déductions que peut faire une conjonction de telles

contraintes est limitée. Par conséquent, nous avons, pour traiter ce problème, introduit un nouvel algorithme de filtrage pour une contrainte automate multi-pondérée : `multicost-regular` [64]. Nous avons montré que cette contrainte permet de faire des déductions plus importantes qu'une simple conjonction de `cost-regular` ou qu'une conjonction de `regular` et `global-cardinality`. Enfin, la contrainte `soft-multicost-regular` a été introduite afin de gérer les fonctions de pénalisations des contraintes automates souples.

Fort de ce bagage théorique, nous allons maintenant, dans la dernière partie de cette thèse, nous rapprocher de la résolution de problèmes de planification de personnel (PSP). Nous y présenterons ce qu'est un PSP, ainsi que les techniques usuelles en programmation par contraintes pour les résoudre. Enfin, une évaluation des améliorations proposées dans cette thèse, y sera présentée.

Troisième partie

Problèmes de planification de personnel : Modélisation et Résolution

Chapitre 8

Les problèmes de planification de personnel

Sommaire

Les problèmes de planification de personnel (PSP), du fait de leur complexité et de l'intérêt qu'ils suscitent au sein des milieux hospitaliers comme industriels ont été beaucoup étudiés en Recherche Opérationnelle et en Intelligence Artificielle. Les PSP connus également sous le nom de « Nurse Rostering Problems (NRP) » (problèmes de planification d'infirmières) consistent à générer un planning associant une tâche ou un quart de travail (par exemple, chirurgie ou quart du matin), à chaque période de temps, un ensemble de tâches, de périodes ou d'équipes (équipe du matin, chirurgie, entrepôt, ...), à du personnel caractérisé par une fonction, des qualifications et sujet à des régulations parfois complexes de leurs horaires. La plupart des PSP sont NP-complets [57] et l'hétérogénéité des règles concernant le personnel en font des problèmes particulièrement difficiles à modéliser et à résoudre. Dans ce chapitre, nous présenterons ce qu'est un PSP et les techniques usuelles en programmation par contraintes pour les résoudre.

8.1 Introduction

Dès lors qu'une structure est organisée, la capacité de placer les bonnes personnes au bon moment devient cruciale pour satisfaire les besoins d'un service, d'une école ou d'une entreprise. On définit les problèmes de planification de personnel comme le procédé consistant à construire de manière optimisée les emplois du temps de travail du personnel. Ce problème, fortement étudié ces dernières années [47] demande généralement d'affecter du personnel suffisamment qualifié à différents services dans le temps,

tout en respectant des règles métiers, résultant des conventions de travail, des lois ou de la spécificité d'un poste. La plupart des approches proposées jusqu'alors en Recherche Opérationnelle et en Intelligence Artificielle se concentrent sur l'efficacité de la résolution et n'abordent pas, ou peu, les problèmes de modélisation. En effet, les approches de Recherche Opérationnelle se présentent souvent sous la forme d'une technique de résolution, dédiée à une instance de problème unique. Il existe néanmoins des approches où la flexibilité du modèle, outre l'efficacité de la méthode de résolution, est mise en avant : dans leur article, Artigues et al. [4] proposent une méthode hybride basée sur de la programmation linéaire en nombres entiers (PLNE) et de la programmation par contraintes pour modéliser et résoudre un problème de planification de production, lié à un problème de planification de personnel. Plus généralement, la contrainte `regular`, développée par Pesant [73] et qui permet de représenter des séquencements à l'aide d'automates finis, a beaucoup facilité la modélisation des règles de travail, de manière flexible.

Dans ce chapitre, nous définirons dans un premier temps ce qu'est un problème de planification de personnel, puis nous parlerons des approches existantes dans les domaines connexes à la programmation par contraintes. Enfin, nous présenterons les méthodes de modélisation et de résolution existantes en programmation par contraintes.

8.2 Terminologie

Comme nous l'avons indiqué dans l'introduction de ce chapitre, la planification de personnel consiste à attribuer à chaque employé, une équipe ou un créneau pendant lequel il doit travailler. Cependant, plusieurs approches existent, et toutes ne considèrent pas par exemple un employé de la même manière. Dans cette section nous définissons les termes que nous emploierons par la suite, en rappelant comment ils sont utilisés dans la littérature.

8.2.1 Personnel

Dans la littérature des problèmes de planification de personnel, il existe en réalité deux types de problèmes [97, 9]. Le premier consiste à trouver le nombre de personnes à employer selon les prévisions d'une demande sur une période. Il faut déterminer les compétences nécessaires pour certaines tâches et calculer la longueur et le début des périodes de travail, de manière à répondre à des profils particuliers de demande. Il s'agit des problèmes dits de « staffing ». Ils ressemblent beaucoup à ce qui se fait dans le cadre des problèmes d'ordonnancement où un ensemble de tâches doivent être positionnées sur un ensemble de machines, de manière à réduire le temps global, pour exécuter toutes les tâches. Le type de problèmes auxquels nous nous intéresserons est un problème d'affectation. Il consiste à générer les emplois du temps du personnel, c'est-à-dire à savoir pour chaque employé, si il travaille ou non à une période donnée. S'il travaille, il faut déterminer à quelle équipe il appartient tout en respectant des contraintes de couverture sur chaque période et des contraintes métiers pour chaque employé. Pour résumer, les premiers problèmes consistent à déterminer le nombre nécessaire d'employés pour réaliser un ensemble de tâches, là où les problèmes d'affectation pour le personnel consistent à déterminer les activités d'un employé, que ce soient des tâches, du repos ou même le déjeuner.

Représentations. Pour modéliser ces derniers problèmes, il faut être capable de modéliser les employés. Deux approches existent : historiquement, les PSP étaient formulés sous forme de problèmes de « staffing » et de ce fait, on cherchait à trouver le nombre d'employés nécessaires. Ainsi dans ces modèles les employés sont anonymes et n'ont pas de caractéristiques spécifiques. Des approches plus orientées « personnel » utilisent dans un premier temps des employés anonymes. Dans leur article, Warner et

Prawda [98] proposent une méthode où un ensemble de séquences ou motifs possibles sont générés pour un service entier. Ils proposent par la suite d'affecter à chacun de ces motifs un employé. Cette méthode ne prend pas en compte les spécificités des employés. Ozkarahan et Bailey [69] proposent une méthode proche dans le sens où ils génèrent un ensemble de motifs d'activités réalisables (i.e. qui respectent les règles légales et les contraintes métiers). C'est ensuite aux infirmières de décider quelle séquence elles préfèrent réaliser.

Plus récemment, les personnels ont été modélisés de manière individuelle dans les systèmes de planification. Cela permet notamment, de prendre en compte les préférences et contraintes spécifiques des employés. De plus, cette approche permet de considérer les compétences individuellement et ainsi modéliser plus fidèlement les besoins d'un service. Par exemple, il existe souvent dans les hôpitaux des chirurgiens seuls aptes à pratiquer certaines opérations. D'un point de vue humain, la communauté de la planification de personnel s'est également rendu compte assez tôt, que la personnalisation des emplois du temps est une condition nécessaire à l'acceptation d'un logiciel de planification [90]. Ainsi, la plupart des logiciels de planification commerciaux, tels ANROM [95], INTERDIP [1] et ORBIS Dienstplan [66], proposent un système de compte individuel où chaque infirmière peut entrer ses préférences et ses contraintes.

Compétences. Choisir du personnel qualifié pour une activité donnée est une des contraintes les plus importantes lorsqu'il s'agit de sélectionner un employé [101, 102]. Il existe de nombreuses façons de caractériser les employés. La plupart du temps, il s'agit avant tout de connaître les compétences, les qualifications et l'expérience de chacun. Plusieurs cas de figure apparaîtront dans les problèmes que nous traiterons :

- les employés ont tous la même qualification ;
- il existe des groupes de personnes de qualification identique ;
- les employés ont plusieurs compétences et qualifications qui leur sont propres.

Selon le cas de figure, le modèle que nous utiliserons évoluera pour tenir compte des qualifications. Il s'agira la plupart du temps de nouvelles contraintes à ajouter au modèle. Certaines activités nécessiteront alors une ou plusieurs compétences particulières empêchant son affectation à certains employés. D'un autre côté, certaines tâches ne nécessitant pas de qualification particulière seront plus coûteuses, si elles sont effectuées par du personnel qualifié, par exemple un chirurgien gardant un malade.

8.2.2 Périodes de travail

Il existe une grande variété de problèmes de planification de personnel, chacun prenant en compte les spécificités de l'entreprise ou de l'organisation concernée. C'est pourquoi il existe de nombreuses façons de représenter un planning. Les problèmes de modélisation sont inévitables, car issus de l'équilibre que l'on cherche à trouver entre la précision et la complexité de la résolution. La représentation usuelle d'un planning est similaire à ce qui se fait lorsqu'il est réalisé à la main. On représente un planning par une matrice à deux dimensions dans laquelle chaque ligne représente le planning d'une seule personne. Les colonnes représentent la division du temps que l'on souhaite : une colonne peut représenter une journée, une période de travail, ou dans certaines applications une activité.

Lorsque l'on parle de planification automatisée, Cheang et al. [35] réferencent trois « vues » :

- la vue personne-jour qui est la représentation naturelle de la planification « à la main » une grille en deux dimensions où chaque case représente l'activité associée à une personne le jour ou la période indiquée dans la colonne ;

- la vue personne-temps où chaque variable représente l'affectation d'une période de temps à une personne ;
- la vue personne-motif où l'on affecte à une personne une séquence d'activités complète.

La première « vue » personne-jour est la plus usitée dans la littérature de la planification de personnel [82, 99, 54, 67]. La deuxième vue, moins utilisée apparaît essentiellement dans les problèmes où la planification se fait en fonction de tâches à exécuter [92] ou dans le domaine des transports [34, 89]. La vue personne-motif offre l'avantage de restreindre énormément l'espace de recherche, il faut néanmoins générer les « motifs » autorisés avant de les affecter aux personnes. Il n'y a alors aucune garantie qu'une solution sera trouvée et il est très difficile de modéliser les préférences en utilisant cette vue [30, 44, 63]. S'il est délicat de dire quelle vue est la meilleure en raison de la nature des problèmes considérés, nous pouvons néanmoins dire à ce niveau que la vue personne-motif n'offrira pas la souplesse nécessaire que nous cherchons pour modéliser un grand nombre de problèmes. Par ailleurs, nous verrons par la suite que la vue personne-temps s'adapte mal à la programmation par contraintes.

Autour de ces modélisations, nous pouvons définir différents types de périodes de travail :

- les périodes « on/off ». Un employé est soit au travail soit en repos. Cela correspond à un modèle où il n'y a qu'une seule activité [31] ;
- les périodes de travail fixées. Les journées de travail sont divisées en différentes périodes. Par exemple dans la plupart des hôpitaux, chaque journée est composée de trois périodes : matin, soir et nuit ;
- les périodes de travail variables. Les journées de travail sont également divisées, mais les dates de début et de fin de chaque période ne sont pas fixées. Ce genre de problèmes est référencé notamment dans l'article de Bailey et Field [7] ;
- les modèles d'affectations de tâches sont quant à eux utilisés pour les problèmes de planification où les compétences et qualifications sont définies pour tous les employés [49, 100]

Dans le cadre de nos travaux, nous nous intéresserons exclusivement aux périodes de travail fixées, et donc par implication aux périodes « on/off ». La principale raison est que ce modèle se fond parfaitement dans le paradigme de la programmation par contraintes. Par ailleurs, les instances que nous avons traitées ne présentent pas d'autres types de périodes.

Taille des plannings. Il est évident que plus un planning est long, plus sa génération est compliquée. Les premières méthodes d'automatisation de la planification étaient de ce fait souvent limitées à quelques jours. Dans les années 70, Warner et Prawda [98] ont généré des plannings sur un horizon de quatre jours. Cependant, il existait déjà des approches où le planning était cyclique, c'est-à-dire qu'il se répétait indéfiniment, on pouvait donc étendre ces quatre jours [25]. La plupart des approches non cycliques s'attaquent à des problèmes dont l'horizon est situé entre une [32, 6] et six semaines [8, 97]. Certains sont même allés jusqu'à planifier un service hospitalier pour un an [22]. D'un point de vue pratique, la plupart des demandes de planning se font pour un mois, soit quatre semaines.

8.2.3 Contraintes

Nous l'avons bien compris, la difficulté de la génération de plannings, dépend non seulement de l'horizon de planning et du nombre de personnes, mais aussi des contraintes qui sont imposées. La difficulté repose d'une part sur l'hétérogénéité des contraintes rencontrées et d'autre part sur le caractère souvent conflictuel de ces contraintes. Dans cette section, nous décrivons les contraintes rencontrées pour les problèmes de planification de personnel, d'un point de vue opérationnel.

Couverture. Du point de vue du management, la principale exigence est de s'assurer qu'il y aura assez de personnels pour couvrir les différentes demandes. Définir les contraintes de couvertures pour un problème de planification de personnel implique donc que l'on connaisse le besoin en personnels et compétences pour chaque période de temps. Définir cette demande est un problème en soit, il faut par exemple connaître le nombre de malades pour déterminer le nombre d'infirmières nécessaires. Nous considérerons toujours que la demande est fixée à l'avance, les méthodes permettant de construire la demande en fonction de données historiques est un problème que nous ne traitons pas dans cette thèse.

Les contraintes de couverture, connaissant la demande, sont définies d'une des manières suivantes :

- **couverture minimale :** le nombre minimum de personnel pour effectuer une opération, ou ayant une qualification donnée.
- **couverture idéale :** le nombre de personnel qu'il faut pour une activité donnée, dans ce cas il peut y avoir parfois plus ou moins de personnes, avec des surcoûts associés.
- **couverture maximale :** le nombre maximum de personnes que l'on peut affecter à une activité. Cette contrainte est rarement explicite, car dans la plupart des problèmes de planification, on cherchera à minimiser le coût du planning, or avoir plus de personnels affectés à une tâche que nécessaire, est très coûteux.

Règles de travail. Nous appellerons « règle de travail », toute contrainte portant sur un employé. Il en existe un grand nombre, voici les principales :

- **horaires travaillés :** il s'agit d'un nombre d'heures minimal ou maximal qu'un employé doit effectuer au cours d'une période donnée. Cette contrainte peut s'appliquer aussi bien sur une période de quelques jours que sur toute la période de planification. Ce type de contraintes est très courant car la plupart des contrats de travail stipulent le nombre d'heures hebdomadaires.
- **repos obligatoires :** le nombre minimum d'heures de repos entre certaines activités ou entre deux périodes de travail.
- **travail consécutif :** le nombre minimum/maximum de jours ou d'activités consécutifs qui doivent être faits avant de pouvoir être au repos. Dans le cadre hospitalier il s'agit souvent de limiter le nombre de quarts de nuit consécutifs effectués.
- **motif d'activités :** les motifs d'activités permettent de modéliser des séquences d'activités interdites ou obligatoires. Il arrive même que le nombre d'occurrences d'un motif soit limité [28, 39]. Ce type de contraintes permet une grande souplesse, car la définition d'un motif est très large. Dans cette thèse, elle sera notre principal intérêt puisque nous verrons que les motifs rationnels, éventuellement munis de compteurs, permettent de modéliser pratiquement toute règle de travail.
- **week-end consécutifs :** cette contrainte permet de limiter le nombre maximum de week-end travaillés consécutivement. Certains problèmes obligent les deux jours du week-end à être soit chômés, soit travaillés [17, 28, 31].

Il existe encore de nombreux types de contraintes concernant les règles de travail. Nous citerons le cas où les vacances doivent être planifiées [29], ou les cas où l'historique des mois précédents doit être pris en compte [28, 59]. Comme indiqué, nous nous concentrerons essentiellement sur la modélisation des motifs d'activités, puisque ce type de contraintes, si modélisé à l'aide d'un outil puissant, permet de représenter un grand nombre de ces règles de travail.

Préférences. Ménager le moral de ses « troupes » est un principe important en management. En effet, avoir du personnel peu motivé au sein d'une organisation a souvent des conséquences dramatiques. Dans le cadre hospitalier, cela se reporte, par exemple, sur la qualité des soins. Dans l'industrie, les délais et les cadences sont plus difficiles à tenir. Un planning généré qui ne prendrait pas en compte les préférences des employés provoquerait, au-delà des risques métiers, un rejet du logiciel de planification.

Les demandes émanant d'employés désireux de ne pas travailler un jour donné, ou préférant faire une activité particulière certains jours sont très présentes dans la littérature. Les méthodes de prise en compte de ces préférences sont diverses, mais toutes mettent en avant l'équité des solutions [3, 48].

Les requêtes portent, la plupart du temps, sur des jours de repos ponctuels. Il peut s'agir parfois de motifs d'activités qu'une personne ne souhaite pas faire. Dans son système de modélisation, Warner [97] propose un questionnaire où les infirmières ont la possibilité de pondérer certaines caractéristiques du planning, comme la possibilité d'avoir un seul jour de repos consécutif ou d'avoir des périodes de travail plus longues, compensées par de plus longs repos. Chaque infirmière disposait de 50 jetons permettant de définir ce qui leur convenait le moins. Vanden Berghe [95] quant à lui propose dans sa thèse un contrat unique par infirmière leur permettant de définir très finement les préférences sur un grand nombre de motifs d'activités.

Contraintes dures VS. contraintes souples. Que ce soit autour des problèmes de planification de personnel, ou n'importe quel autre problème traité en Recherche Opérationnelle ou en Intelligence Artificielle, les contraintes sont souvent exprimées comme étant dures ou souples. Les contraintes dures sont les contraintes qui doivent impérativement être satisfaites, on dit alors que l'emploi du temps est réalisable. Les contraintes souples sont par opposition plus flexibles et autorisent qu'on les ignore. On peut ainsi évaluer la qualité d'un horaire généré tout d'abord par la réalisabilité, à travers les contraintes dures, et par le nombre de contraintes souples violées. L'ajout de poids sur les contraintes permet de modéliser le degré d'importance des règles qu'elles représentent.

La plupart des contraintes de couverture sont considérées comme étant dures, notamment les contraintes de couverture minimale. Cependant, elles peuvent dans certains cas être considérées comme souples lorsque l'on autorise, notamment, à avoir plus de personnel pour une activité et que l'on veut prendre en compte le coût de cette surcharge. De la même manière, les règles de travail peuvent être soit dures, notamment lorsqu'il s'agit de règles légales, telles la durée de travail, ou souples, lorsque l'on considère par exemple, qu'un motif d'activités est fatigant, donc peu conseillé. Enfin, les règles de préférences sont bien évidemment toujours considérées comme souples. La réalisibilité étant parfois même impossible, sans la prise en compte des préférences.

Objectif. La plupart des problèmes de planification de personnel sont des problèmes d'optimisation. La fonction objectif est utilisée pour évaluer la qualité de l'horaire généré [35]. Les premières méthodes de résolution basées sur la programmation linéaire avaient pour objectif de minimiser de manière optimale une fonction objectif souvent complexe. Ces fonctions étant définies de manière complexe pour chaque problème, la souplesse d'une telle approche est pratiquement nulle et il faut reconstruire tout un programme pour traiter un autre problème. De la même façon, les approches heuristiques privilégient les solutions du type un programme pour un problème.

Les approches à base de meta-heuristiques et de programmation par contraintes ont permis une approche plus flexible, en considérant une fonction objectif qui agrège les différents coûts et les pénalités associés à la violation des contraintes.

Les différents types d'objectifs que l'on rencontre dans la littérature sont les suivants :

- minimiser le nombre d'employés [5, 46] ;

- minimiser le coût des employés : par exemple éviter d'utiliser des employés sur-qualifiés [66] ;
- minimiser la violation des contraintes souples à l'aide de fonction de pénalisation [28, 97].

8.3 Approches de résolution en programmation par contraintes

Dans cette section, nous décrirons uniquement les approches de programmation par contraintes pour résoudre des problèmes de planification de personnel. Nous invitons le lecteur qui souhaiterait en savoir plus sur les techniques de résolution des PSP à consulter la bibliographie réalisée en 2004 par Ernst et al. [47]. Très complète, elle comporte près de 700 références.

Comme le montre cette bibliographie, les problèmes de planification de personnel ont été très étudiés et traités à l'aide d'approches très différentes. La programmation par contraintes a notamment été appliquée à ce type de problèmes. Avec INTERDIP [1], Abdennadher et Schenker proposent un système semi-automatisé de planification de personnel dans le cadre hospitalier. Cheng et al. [36] quant à eux intègrent des modèles redondants pour améliorer la génération de l'emploi du temps de la semaine, d'infirmières d'un hôpital de Hong-Kong. Dans ce même cadre, Darmoni et al. [42] ont conçu Horoplan, un logiciel assistant à la création d'emplois du temps pour les hôpitaux, utilisant la programmation par contraintes, pour détecter les conflits et proposer des solutions valides.

Ces trois systèmes utilisent la programmation par contraintes pour la recherche de solutions. Certaines règles de filtrage complexes, spécifiques aux domaines traités ont été intégrées dans ces systèmes, mais l'utilisation de contraintes globales reste limitée.

Dans Hibiscus [23] Bourdais et al. utilisent un ensemble de contraintes pour définir un ensemble de primitives capables de modéliser des règles complexes. On notera notamment l'utilisation des contraintes `stretch` et `pattern` décrites au chapitre 4.

Ces nouvelles contraintes globales, amenant des possibilités de modélisation, sous forme de primitives pour la formulation des problèmes de planification de personnel ont permis de résoudre des problèmes de plus en plus complexes. Néanmoins, l'arrivée de la contrainte `regular` par Pesant [73] en 2004, permettant de contraindre une séquence de variables à respecter un langage rationnel, a permis de modéliser des règles de séquencement encore plus complexes. À l'aide de cette nouvelle primitive, Laporte et Pesant ont proposé un système générique de résolution de PSP [61]. Dans leur article, ils proposent un grand nombre de types de règles pouvant être modélisés de manière triviale à l'aide de leur système.

L'évolution directe de `regular` est la contrainte `cost-regular` introduite par Demassey et al. [43]. Une hybridation de cette contrainte avec de la génération de colonnes leur a permis de résoudre des instances issues d'un magasin de vente au détail et d'une grande banque.

L'extension des contraintes automates aux grammaires hors contextes [75, 56, 86] a permis à Kadioglu et Sellmann d'obtenir d'excellents résultats sur les instances présentées dans l'article de Demassey et al. Néanmoins il faut noter la difficulté de modélisation qu'implique l'utilisation des grammaires hors contextes qui, contrairement aux expressions rationnelles, demandent une maîtrise importante de la théorie des langages.

Quimper et Rousseau [74] présentent une approche de recherche à voisinage large pour résoudre ce même problème. Ils utilisent à la fois grammaires et automates pour modéliser le problème. L'approche génère un ensemble aléatoire de quarts de travail. Chaque quart est ensuite amélioré en évaluant pour chaque activité affectée le gain ou la perte de son remplacement par une autre activité. Un algorithme de programmation dynamique est ensuite utilisé pour trouver le quart qui minimise le coût global. Lorsqu'un minimum local est atteint, de nouveaux quarts sont générés pour tenter d'abaisser ce minimum.

Côté et al. [40, 41] ont introduit une méthode de linéarisation des contraintes `regular` et `context-free grammar`. Ils ont ainsi créé deux modèles pour résoudre les instances de Demassey et al. L'un basé sur la linéarisation de `regular`, `MIP regular`, et l'autre basé sur la contrainte sur les grammaires hors contextes, `MIP grammar`. Les résultats obtenus avec `MIP regular` sont du même ordre que ceux obtenus dans les travaux précédents, mais l'utilisation de `MIP grammar`, bien qu'offrant un pouvoir de modélisation en théorie plus grand, n'apporte pas des résultats satisfaisants sur ce type d'instances.

D'autres types de problèmes, incluant notamment des règles de travail souples, ont été traités à l'aide de la programmation par contraintes. La librairie en ligne du groupe ASAP [70] contient un grand nombre d'instances réelles de problèmes de planification de personnel. Métivier et al. [65] se sont attaqués à certaines de ces instances à l'aide d'un modèle contraintes basé sur leurs contraintes `regular-count` et `cost-regular-count`. Les automates utilisés pour modéliser les règles de séquencement sont construits manuellement. L'utilisation de ces contraintes permet de compter le nombre d'occurrences d'activités ou de motifs d'activités tout en maintenant la cohérence d'arc généralisée pour chaque règle. L'inconvénient de cette méthode est dans la taille de l'automate puisque le nombre d'états des automates construits dépend de la valeur des compteurs utilisés. La recherche de solution est améliorée grandement par l'utilisation de méthodes de voisinage complexes : VNS, LNS et LDS.

Tous les travaux que nous venons de présenter proposent des techniques pour modéliser et résoudre efficacement des problèmes de planification de personnel en programmation par contraintes. Néanmoins, lorsqu'il s'agit d'algorithmes de filtrage, chacun s'attache à l'améliorer pour une personne à planifier. Régin et Gomes ont montré que lorsqu'un problème contient une matrice de contraintes `global-cardinality` [77], il est possible de faire plus de filtrage qu'en les considérant indépendamment. Or, lorsque l'on traite un problème de planification de personnel, il arrive souvent que plusieurs employés soient liés par les mêmes règles. On peut considérer le modèle pour un tel problème comme une matrice où le planning de chaque employé est représenté sur une ligne et la distribution des activités pour chaque période de temps sur une colonne.

Dans cet esprit, Beldiceanu et al. [11] proposent de déduire automatiquement des conditions nécessaires pour les variables de cardinalité présentes sur les colonnes de la matrice (permettant par exemple de borner le nombre d'occurrences d'une activité à un temps donné) à partir des propriétés des mots que peuvent former les automates sur les lignes de la matrice. Ces techniques, dites de double-comptage, ont été appliquées aux instances artificielles de la NSPLib [96], associant des règles de travail dures avec des coûts d'affectation, permettant de réduire considérablement le nombre de backtracks et le temps de recherche sur de nombreuses instances. Elles ont également permis de résoudre certaines instances inaccessibles avec un modèle classique. Cette technique trouve ses limites lorsque chaque employé possède son propre ensemble de règles. De plus, comme dans l'approche de Métivier et al., l'intégration des contraintes de cardinalité directement dans la topologie de l'automate provoque une explosion de la taille de ce dernier.

8.4 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté l'origine et les objectifs associés aux problèmes de planification de personnel. Ils consistent à générer un planning associant un ensemble de tâches, de périodes ou d'équipes (équipe du matin, chirurgie, entrepôt, ...), à du personnel caractérisé par une fonction, des qualifications, et sujet à des régulations parfois complexes de leurs horaires.

Dans un second temps, nous nous sommes intéressés aux techniques usuelles en programmation par contraintes pour les résoudre. Pour résumer l'intérêt de la programmation par contraintes dans ce cadre,

nous citerons Pesant. Dans son article Constraint-based rostering [72], résumant les avancés de la programmation par contraintes comme outil de résolution des problèmes de planification, il conclut en indiquant que la programmation par contraintes est plus que jamais adaptée à la résolution de ce type de problèmes grâce à l'expressivité qu'apportent les contraintes globales et notamment les contraintes automates.

Chapitre 9

Recherche de solutions

Sommaire

Le chapitre précédent présentait une rapide typologie des problèmes de planification de personnel (PSP), ainsi qu'un état de l'art des approches de modélisation en programmation par contraintes (PPC).

Ces travaux démontrent l'intérêt des contraintes automates pour modéliser et combiner les règles de travail dures les plus complexes, car ces dernières se traduisent le plus généralement par la reconnaissance (ou l'interdiction) de motifs rationnels, dans la séquence d'activités d'un employé. En revanche, la relaxation de ces règles nécessite de compter les occurrences d'un motif dans la séquence. L'expressivité d'un automate fini, et de la contrainte `regular`, n'y suffit plus alors, à moins de considérer des automates de taille éventuellement impratiquable [65, 11].

Dans le chapitre 5, nous avons montré comment les automates pondérés permettaient la modélisation de ces règles souples. Dans le chapitre 6, nous avons proposé un algorithme permettant de combiner l'ensemble de ces règles en un unique automate multi-pondéré. Enfin, dans le chapitre 7, nous avons introduit les contraintes globales `multicost-regular` et `soft-multicost-regular`, permettant de traiter ce type d'automates. Dans ce chapitre, nous montrons la mise en application de l'ensemble de ces techniques, à la modélisation et à la résolution de PSP sur-contraints, illustrée par de riches exemples d'instances de Nurse Rostering Problem conçues pour la compétition PATAT 2010. Nous insistons sur l'aspect entièrement systématique de notre approche, depuis la lecture des instances au format XML jusqu'à la configuration du moteur de résolution.

type k	nom	sémantique
R	`assign_request`	ensemble d'affectations interdites
S	`global_shift`	bornes min/max sur le nombre total de jours travaillés
SC	`consecutive_shift`	bornes min/max sur la durée d'un stretch d'activité travaillé
FC	`consecutive_free`	bornes min/max sur la durée des repos
P_j	`pattern_j`	motif j interdit
SW	`worked_WE`	borne max sur le nombre de week-ends travaillés (ie. au moins un jour du week-end travaillé)
CW	`complete_WE`	week-end complet obligatoire (ie. tous les jours travaillés, ou tous les jours repos)
IW	`ident_WE`	week-end homogène obligatoire (ie. une seule activité travaillée permise)
NW	`night_before_WE`	activité N (nuit) interdite la veille d'un week-end non travaillé
WC	`consecutive_WE`	bornes min/max sur le nombre de week-ends travaillés consécutifs

TABLE 9.1 – Les contraintes d'horaire des instances ASAP et PATAT'10.

9.1 Description du problème

Nous considérons ici un problème de conception d'horaires pour un ensemble d'employés $\mathcal{E}$ sur un horizon de temps discretisé $\mathcal{T} = [1, \ldots, T]$. Il s'agit d'affecter à chaque employé et à chaque instant, une activité prise dans un ensemble fini donné $\mathcal{A}$. On distinguera une activité particulière de $\mathcal{A}$, notée R et appelée *repos*. Les autres activités $a \in W = \mathcal{A} \setminus \{R\}$ sont dites *travaillées* ou *quarts*.

L'affectation est sujette à deux catégories de contraintes : les contraintes de couverture, limitant le nombre d'employés affectés à une activité donnée, à un instant donné ; et les contraintes d'horaire, restreignant les séquencements d'activités possibles sur l'horizon de temps, pour un employé donné.

Une contrainte de couverture est définie par deux entiers $\underline{b_{at}}$ et $\overline{b_{at}}$ désignant les nombres d'employés minimal et maximal requis pour une activité $a \in \mathcal{A}$ et un instant $t \in \mathcal{T}$ donnés. On suppose :

$$0 \leq \underline{b_{at}} \leq \overline{b_{at}} \leq |\mathcal{E}|.$$

Les contraintes d'horaire sont de différents types, prédéfinis, et propres à chaque employé. On note $\{C^k, k \in K_e\}$ l'ensemble des contraintes d'horaire auxquelles l'horaire d'un employé $e \in \mathcal{E}$ est soumis. Les différents types de contraintes d'horaire considérés sont répertoriés dans la table 9.1.

En satisfaction, le problème de conception d'horaires consiste donc à construire, pour chaque employé $e \in \mathcal{E}$, une séquence d'activités $X_e = (X_{e1}, \ldots, X_{eT}) \in \mathcal{A}^T$ telles que :

$$\begin{aligned} &\underline{b_{at}} \leq |e \in \mathcal{E}, X_{et} = a| \leq \overline{b_{at}}, && \forall a \in \mathcal{A}, t \in \mathcal{T} \\ &X_e \text{ satisfait } C^k, && \forall k \in K_e, e \in \mathcal{E}. \end{aligned}$$

Dans un contexte sur-contraint, toute contrainte d'horaire C^k est souple et associée à une fonction de pénalité $f^k : \mathbb{Z}_+ \to \mathbb{Z}_+$ mesurant le coût de violation de la contrainte pour toute affectation donnée : le coût de violation d'un horaire X_e sur une contrainte C^k est déterminé par $f^k(y(X_e, C^k))$, où $y(X_e, C^k)$ est un indicateur du nombre de violations de C^k observées dans la séquence X_e, tel que $y(X_e, C^k) = 0$ si et seulement si X_e satisfait C_k. Le problème de conception d'horaires s'exprime alors comme un problème

d'optimisation combinatoire :

$$(P) : \min \sum_{e \in \mathcal{E}} \sum_{k \in K_e} f^k(Y_e^k) \tag{9.1}$$

$$s.t.\ \underline{b_{at}} \leq |\{e \in \mathcal{E}, X_{et} = a\}| \leq \overline{b_{at}}, \qquad \forall a \in \mathcal{A}, t \in \mathcal{T} \tag{9.2}$$

$$Y_e^k = y(X_e, C^k), \qquad \forall k \in K_e, e \in \mathcal{E} \tag{9.3}$$

$$X_{et} \in \mathcal{A} \qquad \forall t, e \in \mathcal{E} \tag{9.4}$$

$$Y_e^k \in [0, +\infty[\qquad \forall k \in K_e, e \in \mathcal{E} \tag{9.5}$$

Nous proposons ici de modéliser (P) en un problème d'optimisation sous contraintes formé, sur ce même encodage X des solutions, par une matrice de contraintes globales :

- pour chaque *colonne* $t \in \mathcal{T}$, correspond aux contraintes (9.2) une unique contrainte `global-cardinality` [80] ;
- pour chaque *ligne* $e \in \mathcal{E}$, correspond aux contraintes (9.3) une unique contrainte `multicost-regular`.

Avant de présenter ce modèle et ses possibles améliorations, nous détaillons le processus de construction d'une contrainte `multicost-regular` pour un employé $e \in \mathcal{E}$ donné, à partir de la spécification de ces contraintes d'horaire.

9.2 Modélisation des contraintes d'horaire souples

9.2.1 Modélisation individuelle des contraintes et des violations

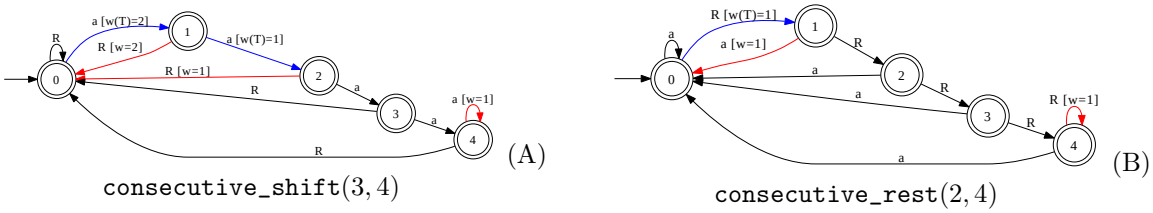
La définition du compteur et du coût de violation d'une contrainte horaire C^k par un horaire X_e dépend du type k de la contrainte. On note ici respectivement $Y_e^k = y(X_e, C^k)$ et $Z_e^k = f^k(y(X_e, C^k))$ ces valeurs. Elles sont définies comme suit :

La contrainte `assign_request`, notée C^R, est définie par une liste d'affectations souhaitées et non souhaitées par l'employé. Ce même formalisme permet de modéliser les souhaits autant que les compétences d'un employé. Chaque affectation $X_{et} = a$ non souhaitée, $(t, a) \in [0, T] \times \mathcal{A}$, est comptabilisée par son degré de non-satisfaction $nw_{ta}^R \in \mathbb{Z}_+$. Si au contraire, l'affectation $X_{et} = a$ est souhaitée, alors le degré de satisfaction $sw_{ta}^R \in \mathbb{Z}_+$ est comptabilisé sur les autres affections possibles de la période. Ainsi, le degré de violation d'une affectation est défini par : $w_{ta}^R = nw_{ta}^R + \sum_{b \neq a} sw_{tb}^R$. On note Y_e^R le compte total des affectations sur l'ensemble des périodes. Le coût de violation associé à cette contrainte est alors égal à ce compte :

$$Z_e^R = Y_e^R, \qquad Y_e^R = \sum_t w_{tX_{et}}^R.$$

La contrainte `global_shift`, notée C^S, impose une borne minimale l_e^S et maximale u_e^S sur le nombre d'activités travaillées dans la séquence X_e. Pour comptabiliser les violations de cette contrainte, on dénombre les activités travaillées dans la séquence X_e par $Y_e^S = \sum_t w_{tX_{et}}^S$, en définissant : $w_{ta}^S = 1$ si a est une activité travaillée, et $w_{tR}^S = 0$ sinon. Si par exemple, la fonction de pénalité f^S est linéaire, on associera le coût de violation :

$$Z_e^S = \max(0, l_e^S - Y_e^S, Y_e^S - u_e^S).$$



(A) `consecutive_shift`$(3, 4)$ (B) `consecutive_rest`$(2, 4)$

FIGURE 9.1 – Exemples d'automates pondérés avec $\mathcal{A} = \{a, R\}$.

Chacune des autres contraintes C^k présentées ci-après peut s'exprimer au moyen d'une expression rationnelle et le compteur Y_e^k se calculer comme la longueur du chemin γ représentant le mot X_e dans un certain automate pondéré $(\Pi^k, (w_t^k)_t)$. γ est la séquence des transitions $(\delta_0, \ldots, \delta_T)$ de Π^k, définie de façon récursive par :

- q_0 est l'état initial de Π^k ;
- δ_t est la transition $\delta^k(q_t, X_e(t), q_{t+1})$ de Π^k depuis l'état q_t vers l'état q_{t+1} étiquetée $X_e(t)$.

Y_e^k est alors défini comme la somme des coûts des transitions de γ : $Y_e^k = \sum_t w_t^k(\delta_t)$. Par exemple :

La contrainte `consecutive_shift`, notée C^{SC}, impose une borne minimale l_e^{SC} et maximale u_e^{SC} sur la longueur de tout stretch dans la séquence X_e, un stretch étant une sous-séquence maximale d'activités travaillées. Pour comptabiliser les violations de cette contrainte, on dénombre par Y_e^{SC}, les ensembles de l éléments dans la séquence formant chaque stretch de longueur $l < l_e^{SC}$ et les ensembles des $(u_e^{SC} - l)$ derniers éléments dans chaque stretch de longueur $l > u_e^{SC}$. Ces différents éléments sont calculables au moyen d'un automate pondéré.

La figure 9.1 (A) présente l'exemple d'un tel automate, sur une instance définie par $\mathcal{A} = \{a, R\}$, $l_e^{SC} = 3$ et $u_e^{SC} = 4$. Cet automate possède cinq états tous finals, dont un état initial noté 0. Les coûts $w_t^{SC}(\delta)$ des transitions δ sont tous nuls sauf pour certaines transitions δ, à certains temps t :

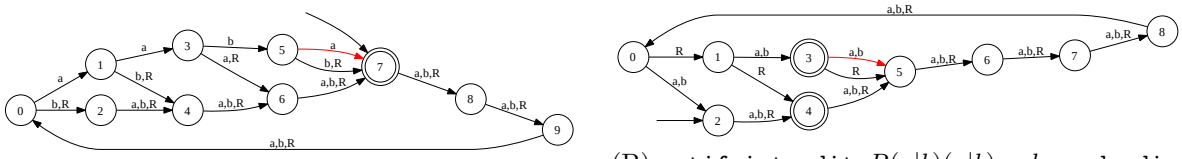
$$\text{(rouge)} : \forall t,\ w_t^{SC}(\delta) = \begin{cases} 2 & \text{si } \delta = (1, R, 0) \\ 1 & \text{si } \delta = (2, R, 0) \\ 1 & \text{si } \delta = (4, R, 4) \end{cases} \quad \text{et} \quad \text{(bleu)} : w_T^{SC}(\delta) = \begin{cases} 2 & \text{si } \delta = (0, a, 1) \\ 1 & \text{si } \delta = (1, a, 2) \end{cases}$$

La transition $\delta = (4, R, 4)$ comptabilise chaque activité travaillée en trop dans tout stretch de durée strictement supérieure à 4. Les autres transitions colorées indiquent la fin d'un stretch de durée l strictement inférieure à 3 et sont pénalisées par le coût $3-l$. La pénalité sur les transitions bleues est comptée uniquement quand la transition est la dernière du chemin δ_T, et correspond au coût d'un stretch terminal de durée insuffisante. Un exemple d'horaire X_e et le compteur-coût $Y_e^{SC} = Z_e^{SC}$ associé :

$X_e =$	a	a	a	a	a	a	a	R	a	R	R	a	a	a	a	R	R	R	a	a	R	R	R	R	a
$y^{SC} = 8 =$					1	1	1			2											1				2

La contrainte `consecutive_rest`, notée C^{FC}, détermine une borne minimale l_e^{FC} et maximale u_e^{FC} sur la durée des repos dans la séquence X_e. Sa variante souple se construit de la même manière que celle de `consecutive_shift`, en inversant simplement les transitions étiquetées par R et celles étiquetées par une activité travaillée.

La figure 9.1 (B) présente l'automate correspondant à $\mathcal{A} = \{a, R\}$, $l_e^{FC} = 3$ et $u_e^{SC} = 4$.



(A) `motif interdit` $aaba$, d=`jeudi`, d_0=`lundi`, d_T=`dimanche`

(B) `motif interdit` $R(a|b)(a|b)$, d=`vendredi`, d_0=`samedi`, d_T=`samedi`

FIGURE 9.2 – Exemples d'automates « motifs à date fixes »

Les contraintes `pattern`, notées C^{P_j}, interdisent l'apparition d'un motif quelconque π_j dans la séquence X_e. On distingue deux types d'interdiction : le motif est soit interdit de manière glissante à tout instant, soit interdit uniquement à partir d'un jour d_j donné de la semaine. Dans les deux cas, on relâche cette interdiction et on modélise le nombre $Y_e^{P_j}$ d'apparitions du motif au moyen d'un automate pondéré. La figure 9.2 illustre cette construction pour deux motifs exemples sur un problème à 3 activités : $\mathcal{A} = \{a, b, R\}$.

Pour l'automate (A) le motif $aaba$ doit être reconnu le jeudi, le planning débutant un lundi et terminant un dimanche. Pour l'automate (B) le motif $R(a|b)(a|b)$ doit être reconnu le vendredi, le planning débutant et terminant un samedi.

La transition rouge correspond à la dernière valeur du motif et n'est accessible qu'après avoir reconnu le motif à partir du jour donné d_j. Les coûts $w_t^{P_j}(\delta)$ sont nuls pour toute transition δ et temps t, sauf $w_t^{P_j}(\delta) = 1$ si δ est la transition rouge et t correspond au dernier jour du motif : $(d_j + l_j)\%7$ avec l_j la longueur du motif.

On notera que la construction n'est valide que si la longueur du motif est inférieure ou égale à 7. En revanche, elle est valide indépendamment des jours de début et de fin du planning : ces jours définissent simplement les états initial et finals de l'automate.

Dans le cas où le motif doit être reconnu de manière glissante, la construction de l'automate est une adaptation de la modélisation de la contrainte `slide` en `regular`.

On notera que toutes les contraintes portant sur les week-ends se modélisent de manière identique une fois exprimé le motif interdit correspondant.

9.2.2 Agrégation des contraintes souples

On construit ainsi, pour chaque contrainte souple C^k, exceptées `assign_request` ($k = R$) et `global_shift` ($k = S$), un automate pondéré $(\Pi^k, (w_t^k)_t)$. La contrainte `cost-regular`$(X_e, Y_e^k, (\Pi^k, w_t^k)_t)$ peut alors être invoquée pour modéliser les contraintes simultanées :

$$X_e \in \mathcal{L}(\Pi^k) \ \wedge Y_e^k = \sum_t w_t^k(\delta_t) \quad \forall e, \forall k,$$

sur les variables-horaires X_e et les variables-compteurs Y_e^k. Ce même modèle peut aussi être utilisé pour $k \in \{R, S\}$ en prenant pour $\Pi^R = \Pi^S$ l'automate complet à un état. Des variables-coûts $(Z_e^k)_k$, sont alors liées aux variables-compteurs par une contrainte définie en fonction du type de k. Le tableau 9.2 récapitule, pour chaque type de contrainte k, l'expression du coût en fonction du compteur et la propriété des coûts $w_t^k(\delta)$, qu'il dépende du temps t ou de l'état (initial) de la transition δ.

On note que pour les contraintes reconnaissant l'apparition d'un motif un certain jour de la semaine, le coût $w_t^{P_j}(\delta)$ est indépendant du temps t. Cependant, par construction de l'automate, chaque transition

k	contrainte	Z^k	temps-dépendant w_t^k	état-dépendant w_t^k
R	`assign_request`	Y^R	oui	non
S	`global_shift`	$\max(0, l^S - Y^S, Y^S - u^S)$	non	non
SC	`consecutive_shift`	Y^{SC}	oui	oui
FC	`consecutive_free`	Y^{FC}	oui	oui
P_j	`pattern_j`	Y^{P_j}	non*	oui

TABLE 9.2 – Synthèse des propriétés des coûts sur différents types de contraintes.

de coût non nul n'est empruntée (et son coût compté) que pour des temps t correspondant à un jour de la semaine donné : le dernier jour du motif.

Afin de maximiser le filtrage, nous proposons à l'instar des motifs dans `regular`, d'agréger l'ensemble de ces automates et coûts en un unique automate pondéré et d'assurer simultanément le filtrage de l'ensemble des compteurs au moyen de la contrainte `multicost-regular` :

$$\texttt{mcr}(X_e, (Y_e^k)_k, (\Pi_e, (w_t^k)_{t,k})).$$

L'automate Π_e est obtenu par intersections successives des automates Π_e^k d'après l'algorithme présenté au chapitre 6.

Le modèle PPC de conception d'horaires pour l'employé e s'écrit :

$$(P_e) Z_e = \sum_k Z_e^k \tag{9.6}$$

$$\texttt{mcr}((X_{et})_t, (Y_e^k)_k, (\Pi_e, (w_e^k)_k) \tag{9.7}$$

$$\texttt{rel}_k(Y_e^k, Z_e^k) \qquad \forall k \tag{9.8}$$

$$X_{et} \in \mathcal{A} \qquad \forall t \tag{9.9}$$

$$Y_e^k \in [0, T], Z_e^k \in [0, +\infty[\qquad \forall k \tag{9.10}$$

$$Z_e \in [0, +\infty[\tag{9.11}$$

9.2.3 Minimisation du coût total de violations

Ici, la majorité des compteurs sont identiques aux coûts de violation associés : $Y^k = Z^k$. À défaut de la `soft-mcr`, nous proposons d'améliorer la qualité de la `mcr` en agrégeant tous ces compteurs en un, le premier $Y^C = Z^C$. En effet, le premier compteur joue un rôle particulier dans la propagation de la contrainte et ce compteur fournit une borne inférieure du coût de violation total :

$$(P_e^*) \min Z_e \tag{9.12}$$

$$Z_e = Z_e^C + Z_e^S \tag{9.13}$$

$$Z_e, Z_e^C \in [0, +\infty[\tag{9.14}$$

$$((X_{et})_t, (Y_e^k)_k, (Z_e^k)_k) \in (P_e) \tag{9.15}$$

Les coûts associés au nouveau compteur Y^C sont définis par :

$$w_t^C(\delta) = w_t^R(\delta) + w_t^{SC}(\delta) + w_t^{FC}(\delta) + \sum_j w_t^{P_j}(\delta), \ \forall t, \forall \delta.$$

Cette agrégation a permis d'améliorer grandement les résultats, par un meilleur filtrage (on agrège les contraintes `mcr` et $Z^C = \sum_{k|Y^k=Z^k} Z^k$), et dans l'heuristique basée sur le plus court chemin de la `mcr` dans la 1ère dimension, qui est plus proche du meilleur horaire pour l'employé.

9.3 Modèle d'optimisation sous contraintes du PSP

Ainsi, le problème de planification de personnel se formule par le modèle d'optimisation sous contraintes suivant :

$$(P) : \min Z \tag{9.16}$$

$$s.t. Z = \sum_e Z_e \tag{9.17}$$

$$\texttt{gcc}((X_{et})_e, (U_{at})_{a\in\mathcal{A}\setminus\{R\}}) \qquad \forall t \tag{9.18}$$

$$\sum_e Y_e^S = \sum_a \sum_t U_{at} \tag{9.19}$$

$$((X_{et})_t, Z_e, Y_e^S) \in (P_e) \qquad \forall e \tag{9.20}$$

$$Z \in [0, +\infty[\tag{9.21}$$

$$U_{at} \in [b_{at}, b_{at}] \qquad \forall a \in \mathcal{A} \setminus \{R\}, \forall t \tag{9.22}$$

La contrainte (9.19) est une contrainte redondante d'égalité entre le nombre total de quarts à couvrir ($\sum_a \sum_t U_{at}$) et la somme du nombre de quarts couverts par chaque employé : ce nombre est donné par la variable-compteur Y_e^S.

9.4 Stratégies de recherche

Le problème sur-contraint (P) est un pur problème d'optimisation : comme pratiquement toutes les contraintes sont souples, il possède un espace de solutions réalisables pratiquement égal à l'espace de recherche, et un espace des solutions optimales, en revanche, négligeable en comparaison. Notre objectif a donc été de parvenir à une solution réalisable de bon coût sans en garantir l'optimalité. Nous nous sommes rapidement orientés vers une recherche locale à voisinage large ou *Large Neighborhood Search* (LNS).

9.4.1 Large Neighborhood Search

À partir d'un ensemble initial de solutions, calculé au moyen d'une recherche arborescente limitée, il s'agit de chercher dans un voisinage de chacune de ces solutions, des solutions améliorantes. Celles-ci sont alors ajoutées à l'ensemble des solutions et le processus est réitéré. Il se termine dans une limite de temps fixée en retournant la meilleure la solution trouvée. Le voisinage d'une solution est construit en dés-instanciant un sous-ensemble d'affectations de la solution courante. Il est exploré, comme dans la résolution initiale, au moyen d'une recherche arborescente limitée.

L'implémentation d'une LNS peut donc se faire aisément au-dessus d'un algorithme de branch-and-bound standard. Elle nécessite alors le développement de deux heuristiques particulières : l'heuristique de branchement classique pour la descente arborescente, et l'heuristique de *débranchement* pour la construction des voisinages.

Les heuristiques que nous avons retenues sont présentées ci-après. Notons tout d'abord que la stratégie de recherche en général a été paramétrée de manière empirique. L'analyse des résultats d'expérimentation

nous a amené à définir 3 objectifs de conception des heuristiques :

Alléger le filtrage des contraintes globales lourdes. Le modèle (P) est basé sur une matrice de contraintes globales de fortes complexités temporelles : `global-cardinality` et surtout `multicost-regular`. Comme nous avons vu dans les expérimentations du chapitre 7, ce choix de modélisation s'avère bénéfique en raison des fortes imbrications et interactions des règles élémentaires du problème, qu'un modèle décomposé manque de voir en début de recherche. Malgré tout, il est nécessaire lors de l'utilisation de contraintes lourdes de ce type, de tenter d'alléger leur exécution en cours de recherche, en exploitant notamment l'incrémentalité de l'algorithme à chaque branchement. Ainsi, afin de neutraliser la complexité temporelle des contraintes `multicost-regular` et `global-cardinality` dans ce modèle, nous avons conçu des heuristiques de choix de variable procédant de proche en proche sur la dimension de temps.

Tout d'abord, dans la descente arborescente, l'heuristique de branchement affecte l'ensemble des employés pour un instant t donné, fixant alors la contrainte `global-cardinality` correspondante, avant de considérer l'instant suivant $t+1$. La notion de séquence, intrinsèque aux contraintes automates, fait qu'une décision prise sur un instant t se propage au sein de `multicost-regular` de manière locale sur les instants voisins $t-1$ et $t+1$.

Cette propriété est également exploitée lors de la construction des voisinages, par l'heuristique de débranchement : celle-ci procède en dés-instanciant l'ensemble des variables d'affectation associées à une période de temps continue. Ainsi, seules les `global-cardinality` correspondant aux instants relâchés doivent être relancées, tandis que toutes les `multicost-regular` sont relâchées, mais seulement de manière locale sur la période. Cette approche s'est avérée plus économique que de relâcher intégralement les `multicost-regular` d'un sous-ensemble d'employés donné.

Exploiter les informations de coût internes aux contraintes d'optimisation. Dans un tel problème d'optimisation, où toute solution est réalisable, il est obligatoire de diriger la recherche suivant l'objectif de minimisation des coûts. Des informations de coût sont justement généralement maintenues en interne des contraintes globales d'optimisation, qui les utilisent lors de la back-propagation des variables de coût vers les variables de décision. En l'occurrence, les contraintes `cost-regular` et `multicost-regular` maintiennent les bornes des coûts des chemins du graphe déployé, autrement dit les coûts des solutions de l'automate. Une telle information avait été exploitée dans l'heuristique de choix de valeur associée à `cost-regular` dans [43] : il s'agissait de sélectionner la valeur d'une variable correspondant au chemin de plus petit coût. Nous avons étendu ce résultat à la contrainte "multi-objective" `multicost-regular`, et également étudié la notion de regret grâce à l'exploitation de l'information de coût des `multicost-regular` par l'heuristique de choix de valeur, lors du branchement. La structure interne de graphe de la contrainte automate renseigne en effet sur le(s) coût(s) d'affectation d'une valeur donnée à une variable de la séquence.

Diversifier la recherche. Par ailleurs, la forte densité des solutions et l'heuristique de branchement orientée optimisation assurent que cette dernière se dirige rapidement vers une solution de bonne qualité, voire la meilleure solution dans son voisinage. Autrement dit, une telle heuristique semble se diriger rapidement (en une descente) vers un optimum local, mais elle peut mettre alors un nombre de backtracks non négligeable à s'échapper de cet optimum. L'intérêt alors est de favoriser la diversification. Nous procédons pour cela de deux manières :

- une politique de redémarrage agressive dans la recherche arborescente (fréquence d'environ 200 nœuds, avec croissance géométrique de 10%), compatible avec la stratégie dynamique de choix de variable : le premier jour considéré par l'heuristique est incrémenté à chaque restart ;

- l'exploration d'un grand nombre de voisinages de tailles variables, nombre compensé par une exploration rapide de ces voisinages.

9.4.2 Heuristiques de branchement

Comme indiqué ci-dessus, notre heuristique de branchement procède "en colonnes" sur la matrice des variables de décision X. Pour un instant t fixé, il s'agit de choisir un employé e, tel que la variable X_{et} ne soit pas encore instanciée, et une activité a à laquelle affecter cette variable. Ces choix sont guidés par les informations de coût calculées lors de la dernière propagation des `multicost-regular` et portant, soit sur la solution (plus court chemin avec multiplicateurs) du meilleur sous-problème lagrangien calculée dans l'algorithme de sous-gradient, soit sur la valeur (dans la première dimension) des plus courts chemins supports de chaque affectation. Plusieurs heuristiques ont ainsi été expérimentées pour le choix de l'employé e. Dans tous les cas, l'activité a associée à l'employé choisi e est la valeur du t-ème arc du plus court chemin dans la contrainte $\texttt{multicost-regular}_e$ correspondante. En cas d'égalité, les activités travaillées sont privilégiées par rapport à l'activité repos.

Coût lagrangien. L'exploitation de la meilleure solution lagrangienne (i.e. un chemin γ_e^λ dans l'automate en couches qui définit un certain horaire valide $(X_{et}^\lambda)_t$) a été expérimentée de quatre façons :

- $Z_e(\lambda)$: choix de l'employé e dont la borne inférieure lagrangienne $Z_e(\lambda)$ (i.e. le coût avec multiplicateurs de γ_e^λ) est minimale.
- $Z_{e,\lambda}^C$: choix de l'employé e dont le coût $Z_{e,\lambda}^C$ de l'horaire $(X_{et}^\lambda)_t$ est minimal. Ce coût se calcule en $O(T)$ par $Z_{e,\lambda}^C = \sum_t w_e^C(X_{et}^\lambda)$.
- $Z_{e,\lambda}$: choix de l'employé e dont le coût $Z_{e,\lambda}$ de l'horaire $(X_{et}^\lambda)_t$ est minimal. Ce coût se calcule en $O(T)$ par $Z_{e,\lambda} = Z_{e,\lambda}^C + Z_{e,\lambda}^S$ avec $Z_{e,\lambda}^S = \max(0, l^S - Y_{e,\lambda}^S, Y_{e,\lambda}^S - u^s)$ et $Y_{e,\lambda}^S = \sum_t w_e^S(X_{et}^\lambda)$. (Note : il n'y a pas de coût Z^{SW} dans la première série d'instances).
- $Z_{e,\lambda} - \underline{Z_e}$: choix de l'employé e dont le « regret » $Z_{e,\lambda} - \underline{Z_e}$ est minimal.

Globalement, les premières solutions fournies par ces heuristiques basées sur la meilleure solution lagrangienne sont plutôt mauvaises. En effet, cette solution est très arbitraire (peu/pas d'itérations du sous-gradient, beaucoup de chemins minimaux dans un graphe, borne égale à 0 pour la plupart des employés) au début de la recherche et l'heuristique n'est donc pas guidée.

Coût dans la première dimension. Afin de contourner ce biais, nous pouvons exploiter une seconde information maintenue dans la `multicost-regular` : l'ensemble des plus courts chemins dans la première dimension $k = C$. Plus exactement, `multicost-regular` maintient la valeur courante des plus courts chemins à chaque nœud du graphe, depuis la source ou jusqu'au puits, et selon chaque dimension. Cette information est calculée à chaque propagation de la contrainte, lorsque chaque compteur est propagé individuellement. Elle permet de mieux approximer la notion de regret : $regret_{e,(t,a)}$ est la différence $Z_{e,(t,a)} - Z_e$ entre le plus petit coût $Z_{e,(t,a)}$ parmi toutes les solutions de (P_e) telles que $X_{et} = a$ et le plus petit coût Z_e parmi toutes les solutions de (P_e). Nous proposons ici de calculer l'approximation suivante :

$$regret_{e,(t,a)}^C = \underline{Z_{e,(t,a)}^C} - \underline{Z_e^C}.$$

$\underline{Z_e^C}$ est la valeur minimale de la variable de coût Z_e^C, tandis que $\underline{Z_{e,(t,a)}^C}$ est calculée en parcourant les arcs supports de (t,a) dans le graphe sous-jacent à `mcr`, et en retenant la valeur du plus court chemin passant par un de ces arcs.

La stratégie de choix de variable consiste alors à sélectionner, pour un jour donné, l'employé e (et le quart a dans le cas de la seconde stratégie de choix de valeur) qui minimise $regret^C_{e,(t,a)}$.

choix de :	e		e, $a \in \mathcal{A}$		e, $a \in \mathcal{A} \setminus \{R\}$	
	first Z	best Z	first Z	best Z	first Z	best Z
$Z_e(\lambda)$	112	86	141	92	104	83
$Z^C_{e,\lambda}$	111	86	141	92	104	83
$Z_{e,\lambda}$	186	158	168	97	102	84
$Z_{e,\lambda} - \underline{Z_e}$			155	99		
$regret^C_{e,(t,a)}$	76	70			76	69

TABLE 9.3 – Comparaison pour différentes heuristiques de choix de valeurs/variables de la première et la meilleure solutions trouvées en 2 minutes sur l'instance PATAT/sprint02

Conclusion : Nous avons conservé le choix de variable (employé) / valeur (activité) basé sur le critère regret minimal $regret^C_{e,(t,a)}$. Nous l'avons encore amélioré en tranchant les cas d'égalité suivant la valeur $\underline{Z^C_{e,(t,a)}}$ du plus court chemin passant par une transition (t,a), puis en cas d'égalité, suivant le nombre d'affectations de quarts $\underline{Y^S_e}$. Le choix du couple (e,a) s'effectue donc selon les critères successifs suivants :

$$regret^C_{e,(t,a)} >> \underline{Z^C_{e,(t,a)}} >> \underline{Y^S_e}. \tag{9.23}$$

On note également avoir obtenu un gain important, grâce à une diversification de la recherche, en démarrant le calcul du meilleur employé, à chaque appel, par l'employé suivant du dernier considéré, plutôt que par l'employé 0.

9.4.3 Heuristiques de voisinage

Tout d'abord, précisons que nous utilisons la même heuristique de branchement dans la recherche initiale et dans l'exploration de chaque voisinage. Concernant la construction des voisinages, différentes stratégies ont été proposées dans la littérature des recherches locales pour les PSP. En dehors des mouvements purement aléatoires, on retiendra l'échange d'activités entre employés et l'échange d'activités entre périodes. Dans le contexte de la recherche locale à voisinage large, ces mouvements s'expriment par, respectivement :

- sélectionner un sous-ensemble d'employés et défixer leurs horaires respectifs ;
- sélectionner un sous-ensemble de périodes et défixer les affections de tous les employés sur ces périodes.

La première approche est défavorable à notre modèle car elle nécessite de repropager intégralement les `multicost-regular` des employés sélectionnés. Même pour deux employés, cette approche semble très coûteuse. La seconde approche au contraire nécessite de repropager toutes les `multicost-regular` mais seulement de manière partielle ; dans chaque contrainte, le graphe déployé des solutions est déjà bien réduit par les pré-affectations de l'horaire. Par ailleurs, comme les règles de séquencement s'appliquent, pour la majorité d'entre elles, sur des périodes de temps courtes, il n'est pas utile de relâcher simultanément des périodes de temps trop espacées. Au contraire, les interactions sont fortes sur les périodes de temps contigues.

Ainsi, nous avons retenu le type de construction de voisinage suivant : un nombre p de périodes est choisi aléatoirement puis appliqué à une solution, en relâchant dans cette dernière l'ensemble des variables X_{et}, $e \in \mathcal{E}$, $t \in [i, i+p \mod T]$, itérativement pour tout $i \in [0, T-1[$.

9.5 Conclusion

Dans ce chapitre, nous avons montré comment les contraintes-automates d'optimisation `multicost-regular` pouvaient être appliquées à la modélisation et à la résolution de PSP type Nurse Rostering Problems (NRP) sur-contraints. Nous avons devisé d'une méthode de résolution incomplète de type LNS entièrement automatisable, depuis la lecture de l'instance à la sortie des solutions. Notre implémentation est libre et open-source, et téléchargeable à `https://github.com/sofdem/chocoETP`. Les parseurs pour différents jeux de test sont disponibles : shoe, NSP, PATAT NRP, ASAP. Les résultats expérimentaux sur les instances PATAT et ASAP sont présentés dans la section suivante.

Des améliorations sont évidemment encore envisageables. Notamment, certaines instances ne peuvent être résolues efficacement si l'utilisateur ne précise pas une borne supérieure suffisamment serrée, ou la limite maximale de pénalité à partir de laquelle une contraire est considérée dure. Dans les deux cas, il s'agirait d'intensifier la recherche de bonnes premières solutions avant de lancer la recherche locale. Il s'agirait par exemple, de brancher sur les coûts de manière destructive pour activer la back-propagation au plus tôt. La limite de pénalité souple pourrait ainsi être, partant d'une valeur nulle (toutes les contraintes sont dures) être progressivement incrémentée jusqu'à ce qu'une solution réalisable soit trouvée.

Enfin, la méthode présentée dans ce chapitre s'applique également à la variante de satisfaction du problème. Si elle n'est plus alors utilisée pour comptabiliser les coûts de violation, la contrainte `multicost-regular` reste utile pour modéliser les compteurs d'activités ou de motifs. Nous avons présenté cette approche, équipée alors d'une autre heuristique de branchement, dans l'article [64].

Chapitre 10

Évaluations

Sommaire

Nous avons dans les chapitres précédents proposé une méthode de construction automatique d'automates pondérés représentant des règles de séquencement, s'adaptant parfaitement au cadre des problèmes de planification de personnel. Puis, nous avons introduit un nouvel algorithme de filtrage pour une contrainte automate multi-pondérée agrégeant plusieurs automates pondérés : `multicost-regular`. Au chapitre 9, nous avons présenté le modèle et les stratégies de recherche adaptées à ces contributions.

Dans ce chapitre nous utilisons le cadre construit pour résoudre divers problèmes issus d'une part de la compétition de planification d'infirmières NRP10 et d'autres part de la librairie de problèmes de planification ASAP [70].

10.1 Instances NRP10

Les instances NRP10 de la compétition de planification d'horaires pour infirmières sont issues d'une session particulière de la conférence PATAT 2010. Elles utilisent un format XML pour décrire un ensemble de personnels, ainsi que les règles qui s'appliquent à chacun d'entre eux. D'autres balises permettent de définir les compétences et la demande de couverture pour chaque activité pour chaque période.

Les instances proposées sont divisées en trois catégories : `sprint`, `medium`, `long`. La différence majeure entre chaque catégorie est le nombre d'employés à planifier : respectivement 10, 30 et 50.

Il existe également deux instances jouets appelés toy1 et toy2.

10.1.1 Format d'instance

Les instances utilisées sont décrites au format XML. Le fichier permet de décrire complètement la période à planifier, les activités autorisées, les différents contrats de travail et leurs règles inhérentes, les

employés associés à un contrat de travail et le cas échéant leurs compétences, les préférences des employés et la demande d'activités journalières.

La période de planification est définie par les deux balises suivantes :

```
<StartDate>2010-01-01</StartDate>
<EndDate>2010-01-28</EndDate>
```

Une activité possède diverses caractéristiques, la balise `Skills` indique notamment l'ensemble des compétences requises par un employé pour effectuer cette activité.

```
<Shift ID="E">
    <StartTime>06:30:00</StartTime>
    <EndTime>14:30:00</EndTime>
    <Description>Early</Description>
    <Skills>
        <Skill>Nurse</Skill>
    </Skills>
</Shift>
```

Le format permet de définir des motifs « Pattern » qui pourront par la suite être forcés ou interdits. Le motif suivant LE représente la succession d'une activité le soir « late » suivie d'une activité le matin « early ».

```
<Pattern ID="3" weight="100">
    <PatternEntries>
        <PatternEntry index="0">
            <ShiftType>L</ShiftType>
            <Day>Any</Day>
        </PatternEntry>
        <PatternEntry index="1">
            <ShiftType>E</ShiftType>
            <Day>Any</Day>
        </PatternEntry>
    </PatternEntries>
</Pattern>
```

Des contrats sont par la suite définis : un contrat sera associé à un employé et représente l'ensemble des règles auxquelles il est sujet. Le contrat suivant représente l'ensemble des règles pour un employé à plein temps. Diverses règles sont représentées au sein de ce contrat. Nous avons associé à chaque contrat une contrainte `multicost-regular`. Chaque règle est ainsi traduite soit par un compteur d'activités souple, soit par un automate. Les motifs interdits (balise « UnwantedPatterns ») sont quant à eux modélisés par un ensemble d'automates dont l'union représente l'ensemble des successions d'activités interdites.

```
<Contract ID="0">
    <Description>fulltime</Description>
    <SingleAssignmentPerDay weight="1">true</SingleAssignmentPerDay>
    <MaxNumAssignments on="1" weight="3">20</MaxNumAssignments>
    <MinNumAssignments on="1" weight="3">10</MinNumAssignments>
    <MaxConsecutiveWorkingDays on="1" weight="4">5</MaxConsecutiveWorkingDays>
    <MinConsecutiveWorkingDays on="1" weight="4">3</MinConsecutiveWorkingDays>
```

```
    <MaxConsecutiveFreeDays on="1" weight="7">4</MaxConsecutiveFreeDays>
    <MinConsecutiveFreeDays on="1" weight="3">3</MinConsecutiveFreeDays>
    <MaxConsecutiveWorkingWeekends on="1" weight="5">3</MaxConsecutiveWorkingWeekends>
    <MinConsecutiveWorkingWeekends on="1" weight="5">2</MinConsecutiveWorkingWeekends>
    <MaxWorkingWeekendsInFourWeeks on="0" weight="1">0</MaxWorkingWeekendsInFourWeeks>
    <WeekendDefinition>SaturdaySunday</WeekendDefinition>
    <CompleteWeekends weight="10">true</CompleteWeekends>
    <IdenticalShiftTypesDuringWeekend weight="10">true</IdenticalShiftTypesDuringWeekend>
    <NoNightShiftBeforeFreeWeekend weight="10">true</NoNightShiftBeforeFreeWeekend>
    <AlternativeSkillCategory weight="10">true</AlternativeSkillCategory>
    <UnwantedPatterns>
        <Pattern>0</Pattern>
        <Pattern>1</Pattern>
        <Pattern>2</Pattern>
        <Pattern>3</Pattern>
        <Pattern>4</Pattern>
        <Pattern>5</Pattern>
        <Pattern>6</Pattern>
        <Pattern>7</Pattern>
        <Pattern>8</Pattern>
    </UnwantedPatterns>
</Contract>
```

Comme indiqué précédemment, un employé possède un contrat et des compétences :

```
<Employee ID="1">
    <ContractID>0</ContractID>
    <Name>1</Name>
    <Skills>
        <Skill>Nurse</Skill>
        <Skill>HeadNurse</Skill>
    </Skills>
</Employee>
```

Les couvertures d'activités par période sont définies par jour de la semaine, en indiquant le nombre d'activités demandées. Dans l'exemple suivant, l'activité du matin doit être effectuée par 7 personnes :

```
<DayOfWeekCover>
    <Day>Wednesday</Day>
    <Cover>
        <Shift>E</Shift>
        <Preferred>7</Preferred>
    </Cover>
</DayOfWeekCover>
```

Finalement, des préférences d'affectations propres à chaque employé peuvent être formulées toujours à l'aide de balises XML spécifiques.

10.1.2 Évaluations

Le framework décrit au chapitre 9 permet de parser ce type de fichier et de créer le modèle contraintes associé dans un format lisible par le solveur de contraintes `CHOCO` [33]. Les contraintes `multicost-regular` et `soft-multicost-regular` ainsi que l'ensemble des heuristiques de recherches sont également implémentés dans `CHOCO`.

Lors de la compétition, les temps de résolution étaient bornés selon le type d'instances : 8 secondes pour les instances sprint, 10 minutes pour les instances médium et 10 heures pour les instances long. Les résultats que nous présentons ne respectent pas ces temps, notamment parce qu'ils sont très pénalisants pour notre approche pour les instances sprint. Nous avons néanmoins résolu les instances en utilisant le framework avec les paramètres suivants :

- restart tous les 1000 nœuds ;
- croissance de la limite de facteur 1.1 par restart ;
- démarrage du LNS après 28 restarts ;
- relaxation de 2, 3, 5, 8 puis 15 colonnes consécutives aléatoirement ;
- problème partiel arrêté au bout de 200 backtracks.

La solution pour la plus grosse instance (long late 5) a ainsi été obtenue en moins de 5 minutes.

Les tableaux 10.1 présentent les résultats obtenus. La colonne initiale indique le coût de violation obtenu après la recherche arborescente au bout de 28 restarts. La colonne LNS indique le score obtenu après la succession de LNS. Enfin la colonne Δ indique la distance relative à l'optimal.

Les résultats obtenus après la recherche initiale sont déjà très bons dans le cas général. Néanmoins, on constate que le LNS améliore grandement les résultats et les rapproches des solutions optimales.

Ce comportement s'explique par le caractère purement optimisation de ces instances. En effet, toutes les contraintes étant souples, toutes affectations des variables de l'horaire forment une solution acceptable. La qualité de la solution initiale est obtenue grâce à l'heuristique basée sur les regrets qui exploite la structure de graphe des contraintes `multicost-regular`.

bench	n	initial	LNS	optimal	Δ
sprint	1	70	59	56	5%
sprint	2	69	62	58	7%
sprint	3	63	60	51	15%
sprint	4	70	60	59	2%
sprint	5	67	63	58	8%
sprint	6	61	61	54	11%
sprint	7	66	61	56	8%
sprint	8	67	63	56	11%
sprint	9	66	60	55	8%
sprint	10	61	**52**	52	0%
sprint late	1	54	50	37	26%
sprint late	2	55	49	42	14%
sprint late	3	63	56	48	14%
sprint late	4	102	88	75	15%
sprint late	5	59	50	44	12%
sprint late	6	47	**42**	42	0%
sprint late	7	55	49	42	14%
sprint late	8	17	**17**	17	0%
sprint late	9	30	27	17	37%
sprint late	10	61	55	43	28%
bench	n	initial	LNS	optimal	Δ
medium	1	256	242	240	1%
medium	2	253	**240**	240	0%
medium	3	246	**236**	236	0%
medium	4	249	243	237	2%
medium	5	327	309	303	2%
medium late	1	273	202	158	22%
medium late	2	32	27	18	33%
medium late	3	52	39	29	26%
medium late	4	73	60	35	42%
medium late	5	184	179	107	40%
bench	n	initial	LNS	optimal	Δ
long	1	207	199	197	1%
long	2	237	220	219	0.5%
long	3	245	**240**	240	0%
long	4	311	310	303	2%
long	5	295	288	284	1%
long late	1	347	275	235	15%
long late	2	372	289	229	21%
long late	3	347	280	220	21%
long late	4	375	307	221	28%
long late	5	167	120	83	31%

TABLE 10.1 – Résultats des évaluations sur les instances NRP10

10.2 ASAP

Les instances ASAP [70] sont également présentées dans un format XML très proche de celui de NRP10. La différence fondamentale réside dans l'absence de règle spécifique dans les contrats. Toutes les règles sont représentées sous forme de motifs dont le nombre d'apparitions est borné. Ainsi pour représenter la règle limitant le nombre de repos pour un employé, la règle sera décrite comme la limitation de l'apparition du motif constitué d'une activité repos.

Si cette représentation facilite la lecture du fichier d'instance, elle complique considérablement la création d'un modèle efficace. En effet, nous considérons pour les instances NRP10 qu'un motif sera représenté par un automate, et si son nombre d'apparitions est limité, par un automate pondéré. Or, lorsqu'il s'agit de compter simplement une activité, l'emploi d'un compteur simple est plus efficace que l'agrégation d'un automate pondéré à un seul état. De plus l'automate représentant les règles sur les fins de semaines peut se construire de diverses manières. La représentation de ces règles par un motif dans les instances ASAP produit un automate non optimisé (n'intégrant pas de compteur). Afin de réduire l'impact de cette modélisation, nous avons ajouté à notre parseur un filtre permettant d'identifier les motifs étant en réalité des compteurs simples ainsi qu'un ensemble de filtres permettant de reconnaître des règles génériques exprimées sous forme de motifs.

Enfin sur certaines instances, de nouvelles règles sont apparues comme la limitation de la consommation de ressources temporelles. Nous ne sommes pas aujourd'hui capables de parser ces règles. Cependant, la modularité de notre framework permettrait de les ajouter relativement rapidement.

Nous nous sommes tout d'abord attaqués aux instances non sur-contraintes de la librairie. Afin de les résoudre nous avons branché sur la variable représentant la violation du problème. Ainsi, lorsqu'elle est fixée à zéro, la back-propagation des contraintes `multicost-regular` permet de rendre « dures » les contraintes du problème.

Mise à part cette modification de la recherche, nous avons utilisé le modèle présenté au chapitre 9 tel quel.

Notre modèle trouve ainsi une solution optimale de coût 0 pour les instances ASAP présentées au tableau 10.2. La colonne $|\mathcal{E}| \times |\mathcal{T}|$ indique les dimensions du problème en nombre d'employés et en jours à planifier.

Instance	$\|\mathcal{E}\| \times \|\mathcal{T}\|$	Temps (s)	#nœuds	#backtracks	Temps (s) (Mét09)
Azaiez	13×28	6.3	4006	5574	*233*
Sintef	24×21	1.4	165	53	-
Millar-2S-1.1	8×12	0.5	29	0	*1*
Millar-2S-1	8×12	0.3	25	0	*1*
Ozkarahan	14×7	0.2	24	5	*1*

TABLE 10.2 – Résultats de notre modélisation sur des instances ASAP

Ces premiers résultats nous indiquent que lorsque le coût de violation est fixé, la résolution est très rapide. Nous avons en effet des temps similaires, voire meilleurs aux temps de référence donnés avec la librairie de problèmes ASAP. Le nombre de backtrack montre que les 4 dernières instances sont en réalité très peu contraintes et que l'heuristique permet de construire presque directement une solution optimale. Seule l'instance Azaiez provoque un nombre important de backtrack. Le solveur trouve en effet beaucoup de solutions de coûts supérieurs à 200 avant de trouver l'optimal à zéro après 2 restarts.

Par la suite, nous nous sommes intéressés aux instances sur-contraintes de la librairie de problèmes. Notre framework n'étant pas capable de lire les instances possédant des compteurs sur la charge de travail provoquée par une activité, cela nous restreint au sous-ensemble d'instances présentées dans le tableau

10.3. La difficulté à trouver une première solution de bonne qualité nous a obligé à débrancher le LNS de la recherche de solution. Nous avons néanmoins laissé les restarts de manière à ce que la colonne choisie au départ par l'heuristique soit incrémentée à chaque restart modulo le nombre de jours à planifier. Pour trois de ces instances nous sommes en mesure de trouver l'emploi du temps générant une violation optimale. Néanmoins, même en laissant un temps conséquent, la preuve d'optimalité n'est jamais réalisée sur ces instances. Concernant les instances LLR et ORTEC01, nous obtenons en réalité 330 pour LLR et 295 pour ORTEC01. Nous avons ensuite lancé la procédure LNS à partir de ces solutions qui ont été améliorées jusqu'à obtenir les valeurs présentées dans le tableau 10.3.

Instance	$\lvert\mathcal{E}\rvert \times \lvert\mathcal{T}\rvert$	Temps (s)	Violations	Optimal	Violations & Temps (Mét09)	
GPost	8×28	75	**5**	5	8	*234*
GPost-B	8×28	625	**3**	3	-	
LLR	27×7	114	320	301	314	*119*
Valouxis	16×28	4879	**20**	20	160	*3780*
ORTEC01	16×31	2920	290	270	-	

TABLE 10.3 – Résultats des instances sur-contraintes d'ASAP

10.3 À propos de la taille de l'automate

Lors de la résolution de ces instances, nous nous sommes posés la question de la taille des automates que nous générions compte tenu du nombre important de règles qui sont agrégées dans ces problèmes. La complexité de l'algorithme de filtrage de `regular` dépend de la taille du graphe déployé, égale dans le pire des cas au produit du nombre de variables par la taille de l'automate. Si un tel algorithme peut sembler inapplicable en théorie sur des automates de trop grandes tailles, nos résultats expérimentaux mettent deux choses en évidence. Tout d'abord, les opérations utilisées pour construire automatiquement un AFD à partir de plusieurs règles tendent à générer un automate déjà partiellement déployé (suite aux intersections) et à réduire le nombre d'états redondants dans les différentes couches (suite à la minimisation). Grâce à cela, le graphe déployé généré pendant la première phase d'initialisation de la contrainte peut être largement plus petit que l'automate passé en paramètre Π. Aussi, la seconde phase d'initialisation réduit considérablement encore le graphe déployé Π_n car de nombreux états finals ne peuvent pas être atteints en exactement n transitions. Le tableau 10.4 donne le nombre de nœuds et d'arcs dans les différents automates construits lors de la construction d'une contrainte `multicost-regular` pour le problème *GPost* avec, dans l'ordre : la somme des tailles des AFD générés pour chacune des règles, la taille de l'AFD Π après intersection et minimisation, l'AFD déployé après les deux phases d'initialisation de la contrainte (phase Avant, puis phase Arrière Π_n).

Contrat	Analyse	$\sum$AFD	Π	Avant	Π_n
Temps complet	#Nœuds	5782	682	411	230
	# Arcs	40402	4768	1191	400
Temps partiel	#Nœuds	4401	385	791	421
	# Arcs	30729	2689	2280	681

TABLE 10.4 – Illustration de la réduction des graphes avant résolution

10.4 Conclusion

Nous avons réalisé dans ce chapitre des expérimentations sur des instances de problèmes de planification de personnel issues des librairies NRP10 et ASAP. Ces librairies nous proposent des instances de différentes tailles, parfois sur-contraintes et également parfois modélisées sous la forme de problèmes d'optimisation pures (pas de contraintes dures). Malgré ces différences de structures, le cadre de résolution basé sur la contrainte `multicost-regular` proposé au chapitre 9 a permis de trouver d'excellentes solutions à toutes ces instances.

Si la preuve d'optimalité n'a été réalisée que pour les instances non sur-contraintes, les résultats approchant les optimaux connus nous confirment que notre système de résolution est efficace tout en conservant une généricité dans la création automatisée du modèle. Le cadre présenté est en effet capable d'être autonome depuis la lecture du fichier d'instance jusqu'à l'affichage d'une solution.

Ces expérimentations mettent par ailleurs en avant l'expressivité des automates et la capacité des *meta-contraintes*-automates à modéliser des règles nombreuses et variées. Il est notamment plus facile de modifier un automate ou de le créer de manière systématique que de multiplier les algorithmes de filtrage pour les diverses contraintes. Les résultats obtenus lorsque la variable de violation est fixée montre que la contrainte `multicost-regular`, grâce à l'agrégation d'un ensemble de règles intrinsèquement liées, permet un filtrage efficace.

Enfin, la qualité des solutions obtenues sur les problèmes pures d'optimisation montre l'intérêt de la construction d'heuristiques s'appuyant sur la structure des contraintes. L'agrégation de nombreuses règles au sein d'une seule contrainte `multicost-regular` facilite par ailleurs ce travail.

Chapitre 11

Conclusion

L'objectif de cette thèse était de proposer un ensemble d'outils pour la modélisation et la résolution de problèmes de planification de personnel. Nous avons volontairement limité notre étude applicative aux problèmes de planification d'infirmières, car ils offrent des règles métiers complexes, à même de montrer la puissance de modélisation des automates.

Tout d'abord, nous avons montré qu'il était possible de modéliser sous forme d'automates pondérés un grand nombre de règles de séquencement. Nous avons proposé, pour chaque type de règles, un algorithme permettant d'automatiser la construction de l'automate. Nous avons proposé un algorithme permettant d'agréger les automates pondérés, pour construire un automate multi-pondéré, représentant le langage des séquencements autorisés par l'ensemble des règles agrégées.

Par la suite, nous avons développé les contraintes `multicost-regular` et `soft-multicost-regular` offrant un filtrage efficace pour les contraintes multi-pondérées. Ces contraintes permettent d'assurer qu'une séquence de variables, représentant par exemple la succession d'activités d'une infirmière, respectera ou pénalisera, pour la version souple, toutes les règles de séquencement représentées par l'automate multi-pondéré.

Grâce à ces contraintes, nous avons proposé un modèle complet en programmation par contraintes pour résoudre les problèmes de planification de personnel. En nous appuyant sur le format XML proposé par le groupe ASAP pour décrire un problème de planification, nous avons développé un framework capable de lire les différentes règles de séquencement et de couverture, et de les traduire dans notre modèle de contraintes.

Afin d'améliorer la résolution, nous avons proposé plusieurs techniques, comme les heuristiques guidées par la structure de la contrainte `multicost-regular` et l'intégration de la recherche à grand voisinage (LNS). Nous avons également essayé de relâcher les contraintes de couverture, à l'aide d'une relaxation lagrangienne. L'idée était de calculer une borne inférieure à la violation du problème, afin d'être capable d'utiliser par la suite, la back-propagation des `multicost-regular`. Si l'approche implémentée a donné des résultats excellents sur des petites instances, le comportement a été différent sur les instances ASAP. Ceci est dû au fait que, malgré l'absence des contraintes de couvertures, trouver la solution optimale au sein d'une `multicost-regular` reste un problème NP-difficile. Lorsque les coûts sont modifiés à l'approche de la borne inférieure, le problème devient extrêmement difficile, et bien que l'optimal soit trouvé très rapidement grâce à l'algorithme de filtrage de `multicost-regular`, la preuve d'optimalité prend un temps trop important, pour permettre d'utiliser cette technique efficacement.

Le modèle pur contraintes présenté au chapitre 9 a été par la suite éprouvé sur des instances issues du monde réel et de la compétition de planification d'infirmières NRP10. Les résultats obtenus, présentés

chapitre 10, sont très encourageants dans la mesure où nous nous comparons à des méthodes développées spécifiquement pour chaque instance, là où nous utilisons un cadre de modélisation et de résolution générique et automatisé. De plus, nous nous sommes attaqués, notamment sur les instances NRP10, à des problèmes d'optimisation purs, où la densité de solutions est très importante. La puissance des contraintes `multicost-regular` et `soft-multicost-regular` réside, au delà de l'aspect modélisation, dans une capacité de filtrage accrue, par rapport à un modèle décomposé. Or, dans le cadre de problèmes d'optimisation purs, il est fait usage uniquement de la back-propagation à partir de la variable de coût globale. Dès lors que cette variable est répartie sur plusieurs contraintes de type `multicost-regular`, le filtrage induit devient malheureusement anecdotique.

Nous obtenons cependant sur ces instances des résultats proches de l'optimal ce qui tend à prouver que l'heuristique s'appuyant sur la structure de `multicost-regular` propose des solutions de qualité. Nous n'avons pas fait de recherche sur les travaux autour des heuristiques s'appuyant sur les informations fournies par les structures internes des contraintes, mais si le principe peut être générifié, il serait un atout important pour un solveur de contraintes.

Un aspect que nous n'avons pas abordé, mais essentiel dans les résultats que nous avons obtenus, est celui de l'implantation des contraintes `multicost-regular` et `soft-multicost-regular`. Nous traitons en effet des automates et des graphes supports de grandes tailles. De plus, de nombreux parcours sont effectués dans les graphes, pour calculer les différents plus courts chemins nécessaires au filtrage de nos contraintes. Différentes versions de `multicost-regular` ont été implantées dans `CHOCO`. Les principes généraux que nous en tirons sont les suivants :

Limiter au maximum l'utilisation du paradigme objet. Bien que l'implantation soit réalisée dans le langage orienté objet java, les allocations de mémoires résultant de l'utilisation d'objets sont très coûteuses, lorsqu'il s'agit de charger des graphes de grande taille. L'approche que nous avons retenue est d'utiliser des structures minimalistes composées essentiellement d'entiers pour représenter les arcs et les nœuds d'un graphe. L'existence d'un arc dans le graphe est ensuite représenté par un simple bit, ce qui permet lors d'un retour arrière de le réinsérer dans le graphe à moindre coût. Les structures dites « backtrackables » sont également coûteuses non seulement en mémoire mais aussi pour les opérations simples telle l'addition. La contrainte `multicost-regular` impose un maintien incrémental des plus courts et longs chemins, le long du graphe support. Il est néanmoins moins coûteux de recalculer les valeurs à restaurer en cas de retour arrière plutôt que de les stocker au cours de la résolution.

Ainsi notre implantation est stable en mémoire, c'est-à-dire qu'après l'initialisation du graphe support, aucune mémoire n'est allouée dans la contrainte `multicost-regular` pendant la résolution. Cela garantit que si l'on est capable de charger le problème en mémoire, la résolution n'engendrera pas de surcoût dû à la création de nouveaux objets. La mémoire consommée viendra uniquement de l'arbre de recherche des solutions.

Au-delà des outils que nous avons développés, nous offrons une réflexion sur la puissance et les limites des contraintes globales. Les contraintes `multicost-regular` et `soft-multicost-regular` résolvent un problème sous-jacent NP-difficile : le problème de plus court chemin sous contraintes de ressources. L'utilisation de la relaxation lagrangienne pour réaliser le filtrage de ces contraintes nous permet d'obtenir de bons temps de calculs , mais ne permet pas de réaliser un filtrage complet. Se pose alors la question de la prochaine étape. Doit-on intégrer un maximum de sous-problèmes, fussent-ils NP-difficiles pour enrichir le catalogue déjà très complet de la programmation par contraintes ?

Au vu de la difficulté de développement et d'intégration de tels algorithmes, il faut être capable d'en mesurer l'apport en terme de résolution, mais aussi en terme de modélisation. La contrainte `multicost-regular` est pour nous une réponse positive à ces deux questions. Les résultats expérimentaux de la contrainte `soft-multicost-regular` nous fournissent une indication de la limite que nous sommes capables d'at-

teindre, en terme d'intégration au sein d'une contrainte globale.

De plus, de nombreux solveurs de contraintes permettent aujourd'hui d'accélérer la recherche en parallèlisant la recherche. Cette recherche accélérée donne un poids moins important aux algorithmes « intelligents » de filtrage. Si lors de nos expérimentations nous avons montré qu'une contrainte `multicost-regular` était plus rapide et plus efficace qu'une conjonction de contraintes `cost-regular`, nous avons noté que le nombre de nœuds parcourus par seconde était très inférieur, du fait de l'importance des calculs effectués dans `multicost-regular` (la complexité algorithmique étant identique). L'algorithme de filtrage de `multicost-regular` n'étant aujourd'hui pas parrallèlisable, il n'est pas dit qu'une recherche massivement parrallèlisée dans un modèle décomposé, ne sera pas à l'avenir plus rapide. Bien sûr le problème étant NP-difficile, la puissance du filtrage restera toujours un argument, quelle que soit la puissance de la machine.

Ce dernier constat indique qu'il faut aujourd'hui repenser les algorithmes de filtrage, et a fortiori les systèmes de contraintes, pour prendre en marche le train de la parallèlisation. Nous avons par ailleurs commencé à parallèliser la recherche des plus courts chemins dans le graphe de la `multicost-regular` avec des résultats très prometteurs.

Bibliographie

[1] S. Abdennadher and H. Schlenker. An Interactive Constraint Based Nurse Scheduler. In *Proceedings of The First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming, PACLP*. Citeseer, 1999.

[2] A.V. Aho and M.J. Corasick. Efficient string matching : an aid to bibliographic search. *Communications of the ACM*, 18(6) :340, 1975.

[3] J.L. Arthur and A. Ravindran. A multiple objective nurse scheduling model. *IIE Transactions*, 13(1) :55–60, 1981.

[4] C. Artigues, M. Gendreau, and L.M. Rousseau. A flexible model and a hybrid exact method for integrated employee timetabling and production scheduling. *Practice and Theory of Automated Timetabling VI*, pages 67–84, 2007.

[5] C.S. Azmat, T. H
"urlimann, and M. Widmer. Mixed integer programming to schedule a single-shift workforce under annualized hours. *Annals of Operations Research*, 128(1) :199–215, 2004.

[6] J. Bailey. Integrated days off and shift personnel scheduling. *Computers & Industrial Engineering*, 9(4) :395–404, 1985.

[7] J. Bailey and J. Field. Personnel scheduling with flexshift models. *Journal of Operations Management*, 5(3) :327–338, 1985.

[8] J.F. Bard and H.W. Purnomo. Preference scheduling for nurses using column generation* 1. *European Journal of Operational Research*, 164(2) :510–534, 2005.

[9] N. Beaumont. Scheduling staff using mixed integer programming. *European Journal of Operational Research*, 98(3) :473–484, 1997.

[10] N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing the cardinality-path constraint family. *Logic Programming*, pages 59–73, 2001.

[11] N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. On matrices, automata, and double counting. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 10–24, 2010.

[12] N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global Constraint Catalog. Technical report, EMN, 2008.

[13] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[14] Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraint checkers. *Constraints*, 10(4) :339–362, 2005. ISSN 1383-7133.

[15] Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4) :459–489, 2008.

[16] C. Berge. *Théorie des graphes et ses applications*. Dunod Paris, 1958.

[17] I. Berrada, J.A. Ferland, and P. Michelon. A multi-objective approach to nurse scheduling with both hard and soft constraints. *Socio-Economic Planning Sciences*, 30(3) :183–193, 1996.

[18] C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. Among, common and disjoint constraints. *Recent Advances in Constraints*, pages 29–43, 2006.

[19] C. Bessiere and J.C. Régin. Mac and combined heuristics : Two reasons to forsake fc (and cbj) on hard problems. In *Principles and Practice of Constraint Programming—CP96*, pages 61–75. Springer, 1996.

[20] Christian Bessière. Constraint propagation. Technical Report 06020, LIRMM CNRS/University of Montpellier, March 2006.

[21] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. Reformulating global constraints : The SLIDE and REGULAR constraints. In *SARA*, pages 80–92, 2007.

[22] J.T. Blake and M.W. Carter. A goal programming approach to strategic resource allocation in acute care hospitals. *European Journal of Operational Research*, 140(3) :541–561, 2002.

[23] S. Bourdais, P. Galinier, and G. Pesant. HIBISCUS : A Constraint Programming Application to Staff Scheduling in Health Care. In *Principles and Practice of Constraint Programming (CP'2003)*, volume 2833 of *LNCS*, pages 153–167. Springer-Verlag, 2003.

[24] S. Boyd, L. Xiao, and A. Mutapcic. Subgradient methods. *lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004, 2003.

[25] DJ Bradley and JB Martin. Continuous personnel scheduling algorithms : a literature review. *Journal of the Society for Health Systems*, 2(2) :8, 1991.

[26] Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter Stuckey, and Toby Walsh. Encodings of the sequence constraint. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 210–224. Springer Berlin / Heidelberg, 2007.

[27] D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4) :251–256, 1979.

[28] E. Burke, P. Cowling, P. De Causmaecker, and G.V. Berghe. A memetic approach to the nurse rostering problem. *Applied Intelligence*, 15(3) :199–214, 2001.

[29] E.K. Burke, P. De Causmaecker, S. Petrovic, and G.V. Berghe. Fitness evaluation for nurse scheduling problems. In *Proceedings of Congress on Evolutionary Computation, CEC2001, Seoul, IEEE Press*, pages 1139–1146. Citeseer, 2001.

[30] E.K. Burke, G. Kendall, and E. Soubeiga. A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9(6) :451–470, 2003.

[31] RN Burns. Manpower scheduling with variable demands and alternate weekends off. *Infor*, 16(2) :101–111, 1978.

[32] X. Cai and KN Li. A genetic algorithm for scheduling staff of mixed skills under multi-criteria* 1. *European Journal of Operational Research*, 125(2) :359–369, 2000.

[33] Hadrien Cambazard, Narendra Jussien, François Laburthe, and Guillaume Rochart. The choco constraint solver. In *INFORMS Annual meeting*, Pittsburgh, PA, USA, November 2006.

[34] A. Caprara, P. Toth, D. Vigo, and M. Fischetti. Modeling and solving the crew rostering problem. *Operations Research*, 46(6) :820–830, 1998.

[35] B. Cheang, H. Li, A. Lim, and B. Rodrigues. Nurse rostering problems–a bibliographic survey. *European Journal of Operational Research*, 151(3) :447–460, 2003.

[36] BMW Cheng, J.H.M. Lee, and JCK Wu. A constraint-based nurse rostering system using a redundant modeling approach. In *ictai*, page 140. Published by the IEEE Computer Society, 1996.

[37] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3) :113–124, 1956.

[38] N. Chomsky. *Syntactic structures*. Walter de Gruyter, 2002.

[39] A.H.W. Chun, S.H.C. Chan, G.P.S. Lam, F.M.F. Tsang, J. Wong, D.W.M. Yeung, et al. Nurse rostering at the hospital authority of Hong Kong. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 951–956. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2000.

[40] Marie-Claude Côté, Bernard Gendron, and Louis-Martin Rousseau. Modeling the regular constraint with integer programming. In *Proceedings of the 4th international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR '07, pages 29–43, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] M.C. Côté, B. Gendron, C.G. Quimper, and L.M. Rousseau. Formal languages for integer programming modeling of shift scheduling problems. *Constraints*, pages 1–23, 2007.

[42] S.J. Darmoni, A. Fajner, N. Mahe, A. Leforestier, M. Vondracek, O. Stelian, and M. Baldenweck. Horoplan : computer-assisted nurse scheduling using constraint-based programming. *Journal of the Society for Health Systems*, 5(1) :41–54, 1995.

[43] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4) :315–333, 2006.

[44] KA Dowsland and JM Thompson. Solving a nurse scheduling problem with knapsacks, networks and tabu search. *Journal of the Operational Research Society*, 51(7) :825–833, 2000.

[45] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer-Verlag New York Inc, 2009.

[46] F.F. Easton and N. Mansour. A distributed genetic algorithm for employee staffing and scheduling problems. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 367. Morgan Kaufmann Publishers Inc., 1993.

[47] AT Ernst, H. Jiang, M. Krishnamoorthy, B. Owens, and D. Sier. An annotated bibliography of personnel scheduling and rostering. *Annals of Operations Research*, 127(1) :21–144, 2004.

[48] P. Eveborn and M. R
"onnqvist. Scheduler–A system for staff planning. *Annals of Operations Research*, 128(1) :21–45, 2004.

[49] P. Franses and G. Post. Personnel scheduling in laboratories. *Practice and Theory of Automated-Timetabling IV*, pages 113–119, 2003.

[50] S. Golomg and L. Baumert. Backtrack programming. *Journal of the ACM*, 12 :516–524, 1965.

[51] G. Handler and I. Zang. A dual algorithm for the restricted shortest path problem. *Networks*, 10 :293–310, 1980.

[52] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3) :263–313, 1980.

[53] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education. Addison-Wesley, 2001.

[54] A. Jan, M. Yamamoto, and A. Ohuchi. Evolutionary algorithms for nurse scheduling problem. In *Proceedings of the 2000 congress on evolutionary computation*, pages 196–203. Citeseer, 2000.

[55] Z. Jiang, O. De Vel, and B. Litow. Unification and extension of weighted finite automata applicable to image compression. *Theoretical Computer Science*, 302(1-3) :275–294, 2003.

[56] Serdar Kadioglu and Meinolf Sellmann. Efficient context-free grammar constraints. In *AAAI*, pages 310–316, 2008.

[57] R.M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 2010.

[58] S.C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, 34 :3–41, 1956.

[59] M.M. Kostreva and K.S.B. Jennings. Nurse scheduling on a microcomputer. *Computers & Operations Research*, 18(8) :731–739, 1991.

[60] Werner Kuich and Arto Salomaa. *Semirings, Automata, Languages*. Springer-Verlag, 1986.

[61] G. Laporte and G. Pesant. A general multi-shift scheduling system. *Journal of the Operational Research Society*, 55(11) :1208–1217, 2004.

[62] M. Maher, N. Narodytska, C.-G. Quimper, and T. Walsh. Flow-Based Propagators for the *sequence* and Related Global Constraints. In *Proceedings of CP'2008*, volume 5202 of *LNCS*, pages 159–174, 2008.

[63] C. Maier-Rothe and H.B. Wolfe. Cyclical scheduling and allocation of nursing staff. *Socio-Economic Planning Sciences*, 7(5) :471–487, 1973.

[64] Julien Menana and Sophie Demassey. Sequencing and counting with the `multicost-regular` constraint. In *6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 178–192, Pittsburgh, USA, May 2009. Springer-Verlag.

[65] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni. Solving nurse rostering problems using soft global constraints. *Principles and Practice of Constraint Programming - CP 2009*, pages 73–87, 2009.

[66] A.U.F.M.H. MEYER et al. Solving rostering tasks as constraint optimization. *Lecture notes in computer science*, pages 191–212, 2001.

[67] H.H. Millar and M. Kiragu. Cyclic and non-cyclic scheduling of 12 h shift nurses by network programming. *European Journal of Operational Research*, 104(3) :582–592, 1998.

[68] M. Mohri and F. Pereira Michael. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1) :69–88, 2002.

[69] I. Ozkarahan and J.E. Bailey. Goal programming model subsystem of a flexible nurse scheduling support system. *IIE transactions*, 20(3) :306–316, 1988.

[70] Personnel Scheduling Data Sets and Benchmarks. `http://www.cs.nott.ac.uk/ tec/NRP/`.

[71] G. Pesant. A filtering algorithm for the stretch constraint. In *Principles and Practice of Constraint Programming—CP 2001*, pages 183–195. Springer, 2001.

[72] G. Pesant. Constraint-based rostering. In *The 7th International Conference on the Practice and Theory of Automated Timetabling, PATAT*, 2008.

[73] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'2004*, pages 482–495, 2004.

[74] C.G. Quimper and L.M. Rousseau. A large neighbourhood search approach to the multi-activity shift scheduling problem. *Journal of Heuristics*, 16(3) :373–392, 2010.

[75] Claude-Guy Quimper and Toby Walsh. Decomposing global grammar constraints. *Principles and Practice of Constraint Programming –CP 2007*, pages 590–604, 2007.

[76] P. Refalo. Impact-based search strategies for constraint programming. *Principles and Practice of Constraint Programming–CP 2004*, pages 557–571, 2004.

[77] J.C. Régin and C. Gomes. The cardinality matrix constraint. *Principles and Practice of Constraint Programming–CP 2004*, pages 572–587, 2004.

[78] J.C. Régin and J.F. Puget. A filtering algorithm for global sequencing constraints. *Principles and Practice of Constraint Programming-CP97*, pages 32–46, 1997.

[79] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94 : Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[80] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on AI (AAAI/IAAI'96)*, volume 1, pages 209–215, Portland, August 1996.

[81] G. Richaud, H. Cambazard, B. O'Sullivan, and N. Jussien. Automata for nogood recording in constraint satisfaction problems. *Integration of SAT and CP Techniques*, page 113, 2006.

[82] C.K. Riesbeck and R.C. Schank. *Inside case-based reasoning*. Lawrence Erlbaum, 1989.

[83] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier Science Ltd, 2006.

[84] J. Sakarovitch. *Eléments de théorie des automates*. Vuibert, Paris, 2003.

[85] Klaus U. Schulz and Tomek Miko Lajewski. Between finite state and prolog : constraint-based automata for efficient recognition of phrases. *Nat. Lang. Eng.*, 2 :365–366, December 1996.

[86] M. Sellmann. The theory of grammar constraints. *Principles and Practice of Constraint Programming-CP 2006*, pages 530–544, 2006.

[87] Meinolf Sellmann. Theoretical foundations of CP-based lagrangian relaxation. *Principles and Practice of Constraint Programming –CP 2004*, pages 634–647, 2004.

[88] NZ Shor, KC Kiwiel, and A Ruszcayǹski. Minimization methods for non-differentiable functions. *Springer-Verlag New York, Inc.*, 1985.

[89] D. Teodorovi and P. Lui. A fuzzy set theory approach to the aircrew rostering problem. *Fuzzy sets and systems*, 95(3) :261–271, 1998.

[90] J.M. Tien and A. Kamiyama. On manpower scheduling algorithms. *Siam Review*, 24(3) :275–287, 1982.

[91] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1) :230, 1937.

[92] G.L. Vairaktarakis, X. Cai, and C.Y. Lee. Workforce planning in synchronous production systems* 1. *European Journal of Operational Research*, 136(3) :551–572, 2002.

[93] W.-J. van Hoeve, G. Pesant, L.-M. Rousseau, and A. Sabharwal. Revisiting the *sequence* Constraint. In *Proceedings of CP'2006*, volume 4204 of *LNCS*, pages 620–634, 2006.

[94] Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming : Flow-based soft global constraints. *J. Heuristics*, 12(4-5) :347–373, 2006.

[95] G. Vanden Berghe. *An advanced model and novel meta-heuristic solution methods to personnel scheduling in healthcare*. PhD thesis, Univeristy of Gent, 2002.

[96] M. Vanhoucke and B. Maenhout. On the characterization and generation of nurse scheduling problem instances. *European Journal of Operational Research*, 196(2) :457–467, 2009.

[97] D.M. Warner. Scheduling nursing personnel according to nursing preference : A mathematical programming approach. *Operations Research*, 24(5) :842–856, 1976.

[98] D.M. Warner and J. Prawda. A mathematical programming model for scheduling nursing personnel in a hospital. *Management Science*, 19(4) :411–422, 1972.

[99] G. Weil, K. Heus, P. Francois, M. Poujade, F. de Med, and G. IMAG. Constraint programming for nurse scheduling. *IEEE Engineering in Medicine and Biology Magazine*, 14(4) :417–422, 1995.

[100] C. White and G. White. Scheduling doctors for clinical training unit rounds using tabu optimization. *Practice and Theory of AutomatedTimetabling IV*, pages 120–128, 2003.

[101] H. Wolfe and J.P. Young. Staffing the nursing unit : Part I. Controlled variable staffing. *Nursing Research*, 14(3) :236, 1965.

[102] H. Wolfe and J.P. Young. Staffing the nursing unit : Part II. The multiple assignment technique. *Nursing Research*, 14(4) :299, 1965.

Automates et programmation par contraintes pour la planification de personnel

Julien Menana

Mots-clés : Automates, Planification de personnel, Programmation par contraintes, Relaxation lagrangienne, `multicost-regular`**,** `soft-multicost-regular`

Dès lors qu'une structure est organisée, la capacité de placer les bonnes personnes au bon moment devient cruciale pour satisfaire les besoins d'un service, d'une école ou d'une entreprise. On définit les problèmes de planification de personnel comme le procédé consistant à construire de manière optimisée les emplois du temps de travail du personnel. La motivation de cette thèse est de proposer un moyen d'exprimer ces problèmes de manière simple et automatique, sans avoir à intervenir ensuite dans le processus de résolution. Pour cela, nous proposons de rassembler le pouvoir de modélisation des automates avec la capacité de la programmation par contraintes à résoudre efficacement des problèmes complexes. Le caractère expressif des automates est ainsi utilisé pour modéliser des règles de séquencement complexes. Puis, afin d'intégrer ces automates multi-pondérés dans un modèle de programmation par contraintes, nous introduisons un nouvel algorithme de filtrage basé sur la relaxation lagrangienne : `multicost-regular`. Nous présentons également une version souple de cette contrainte permettant de pénaliser les violations d'une règle modélisée par un tel automate : `soft-multicost-regular`. Le modèle contraintes basé sur ces contraintes est automatiquement construit. Il est résolu à l'aide de la librairie de contraintes CHOCO et a été testé sur des instances réalistes issues des librairies ASAP et NRP10. La recherche de solution est améliorée par l'utilisation d'heuristiques spécifiques basées sur les regrets et s'appuyant sur la structure des contraintes `multicost-regular` et `soft-multicost-regular`.

Automata and Constraint Programming for Personnel Scheduling Problems

Julien Menana

Keywords : Automata, Personnel scheduling problems, Constraint programming, Lagrangian relaxation, `multicost-regular`**,** `soft-multicost-regular`

As soon as a structure is organized, the ability to put the right people at the right time is critical to satisfy the need of a department, a school or a company. We define personnel scheduling problems as the process of building, in an optimized manner, the personnel schedules. The aims of this thesis are to propose a mean to express those problems in a simple and automatic way, avoiding the user to interact with the technical aspects of the resolution. For that matter, we propose to mix the modeling power of automata with the efficiency and modularity of constraint programming for complex problem solving. Thus, we use the expressiveness of the finite multi-valued automata to model complex scheduling rules. Then, to make use of those built automata, we introduce a new filtering algorithm for multi-valued finite automata based on Lagrangian relaxation : `multicost-regular`. We also introduce a soft version of this constraint that has the ability to penalize violated rules defined by the automaton : `soft-multicost-regular`. The constraint model is automatically built. It is solved using the constraint library CHOCO and the whole modeling-solving process has been tested on realistic instances from ASAP and NRP10 libraries. The solution search is finally improved using specialized regret-based heuristics using the structure of `multicost-regular` and `soft-multicost-regular`.